Information Security and Cryptography

Nigel P. Smart

Cryptography Made Simple

Springer

Information Security and Cryptography

Series Editors

David Basin
Kenny Paterson

Advisory Board

Michael Backes
Gilles Barthe
Ronald Cramer
Ivan Damgård
Andrew D. Gordon
Joshua D. Guttman
Christopher Kruegel
Ueli Maurer
Tatsuaki Okamoto
Adrian Perrig
Bart Preneel

More information about this series at http://www.springer.com/series/4752

Nigel P. Smart

Cryptography Made Simple

Springer

Nigel P. Smart
University of Bristol
Bristol, UK

ISSN 1619-7100 ISSN 2197-845X (electronic)
Information Security and Cryptography
ISBN 978-3-319-21935-6 ISBN 978-3-319-21936-3 (eBook)
DOI 10.1007/978-3-319-21936-3

Library of Congress Control Number: 2015955608

Springer Cham Heidelberg New York Dordrecht London
© Springer International Publishing Switzerland 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

This is a reworking of my earlier book "Cryptography: An Introduction" which has been available online for over a decade. In the intervening years there have been major advances and changes in the subject which have led me to revisit much of the material in this book. In the main the book remains the same, in that it tries to present a non-rigorous treatment of modern cryptography, which is itself a highly rigorous area of computer science/mathematics. Thus the book acts as a stepping stone between more "traditional" courses which are taught to undergraduates around the world, and the more advanced rigorous courses taught in graduate school.

The motivation for such a bridging book is that, in my view, the traditional courses (which deal with basic RSA encryption and signatures, and perhaps AES) are not a suitable starting point. They do not emphasize the importance of what it means for a system to be secure; and are often introduced into a curriculum as a means of demonstrating the applicability of mathematical theory as opposed to developing the material as a subject in its own right. However, most undergraduates could not cope with a full-on rigorous treatment from the start. After all one first needs to get a grasp of basic ideas before one can start building up a theoretical edifice.

The main differences between this version and the Third Edition of "Cryptography: An Introduction" is in the ordering of material. Now security definitions are made central to the discussion of modern cryptography, and all discussions of attacks and weaknesses are related back to these definitions. We have found this to be a good way of presenting the material over the last few years in Bristol; hence the reordering. In addition many topics have been updated, and explanations improved. I have also made a number of the diagrams more pleasing to the eye.

Cryptography courses are now taught at all major universities; sometimes these are taught in the context of a Mathematics degree, sometimes in the context of a Computer Science degree, and sometimes in the context of an Electrical Engineering degree. Indeed, a single course often needs to meet the requirements of all three types of students, plus maybe some from other subjects who are taking the course as an "open unit". The backgrounds and needs of these students are different; some will require a quick overview of the algorithms currently in use, whilst others will want an introduction to current research directions. Hence, there seems to be a need for a textbook which starts from a low level and builds confidence in students until they are able to read the texts mentioned at the end of this Preface.

The background I assume is what one could expect of a third or fourth year undergraduate in computer science. One can assume that such students have already met the basics of discrete mathematics (modular arithmetic) and a little probability. In addition, they will have at some point done (but probably forgotten) elementary calculus. Not that one needs calculus for cryptography, but the ability to happily deal with equations and symbols is certainly helpful. Apart from that I introduce everything needed from scratch. For those students who wish to dig into the mathematics a little more, or who need some further reading, I have provided an appendix which covers most of the basic algebra and notation needed to cope with modern cryptosystems.

It is quite common for computer science courses not to include much of complexity theory or formal methods. Many such courses are based more on software engineering and applications of computer science to areas such as graphics, vision or artificial intelligence. The main goal of such courses is to train students for the workplace rather than to delve into the theoretical aspects of

the subject. Hence, I have introduced what parts of theoretical computer science I need, as and when required.

I am not mathematically rigorous at all steps, given the target audience, but aim to give a flavour of the mathematics involved. For example I often only give proof outlines, or may not worry about the success probabilities of many of the reductions. I try to give enough of the gory details to demonstrate why a protocol or primitive has been designed in a certain way. Readers wishing for a more in-depth study of the various points covered or a more mathematically rigorous coverage should consult one of the textbooks or papers in the Further Reading sections at the end of each chapter.

On the other hand we use the terminology of groups and finite fields from the outset. This is for two reasons. Firstly, it equips students with the vocabulary to read the latest research papers, and hence enables students to carry on their studies at the research level. Secondly, students who do not progress to study cryptography at the postgraduate level will find that to understand practical issues in the "real world", such as API descriptions and standards documents, a knowledge of this terminology is crucial. We have taken this approach with our students in Bristol, who do not have any prior exposure to this form of mathematics, and find that it works well as long as abstract terminology is introduced alongside real-world concrete examples and motivation.

I have always found that when reading protocols and systems for the first time the hardest part is to work out what is public information and which information one is trying to keep private. This is particularly true when one meets a public key encryption algorithm for the first time, or one is deciphering a substitution cipher. Hence I have continued with the colour coding from the earlier book. Generally speaking items in red are secret and should never be divulged to anyone. Items in blue are public information and are known to everyone, or are known to the party one is currently pretending to be.

For example, suppose one is trying to break a system and recover some secret message m; suppose the attacker computes some quantity b. Here the red refers to the quantity the attacker does not know and blue refers to the quantity the attacker does know. If one is then able to write down, after some algebra,

$$b = \cdots = m,$$

then it is clear something is wrong with our cryptosystem. The attacker has found out something he should not. This colour coding will be used at all places where it adds something to the discussion. In other situations, where the context is clear or all data is meant to be secret, I do not bother with the colours.

To aid self-study each chapter is structured as follows:

- A list of items the chapter will cover, so you know what you will be told about.
- The actual chapter contents.
- A summary of what the chapter contains. This will be in the form of revision notes: if you wish to commit anything to memory it should be these facts.
- Further Reading. Each chapter contains a list of a few books or papers from which further information can be obtained. Such pointers are mainly to material which you should be able to tackle given that you have read the prior chapter.

There are no references made to other work in this book; it is a textbook and I did not want to break the flow with references to this, that and the other. Therefore, you should not assume that ANY of the results in this book are my own; in fact NONE are my own. Those who wish to obtain pointers to the literature should consult one of the books mentioned in the Further Reading sections.

The book is clearly too large for a single course on cryptography; this gives the instructor using the book a large range of possible threads through the topics. For a traditional cryptography course within a Mathematics department I would recommend Chapters 1, 2, 3, 7, 11, 12, 13, 14, 15, 16

and 17. For a course in a Computer Science department I would recommend Chapters 1, 11, 12, 13, 14, 15 and 16, followed by a selection from 18, 19, 20, 21 and 22. In any course I *strongly* recommend the material in Chapter 11 should be covered. This is to enable students to progress to further study, or to be able to deal with the notions which occur when using cryptography in the real world. The other chapters in this book provide additional supplementary material on historical matters, implementation aspects, or act as introductions to topics found in the recent literature.

Special thanks go to the following people (whether academics, students or industrialists) for providing input over the years on the various versions of the material: Nils Anderson, Endre Bangerter, Guy Barwell, David Bernhard, Dan Bernstein, Ian Blake, Colin Boyd, Sergiu Bursuc, Jiun-Ming Chen, Joan Daemen, Ivan Damgård, Gareth Davies, Reza Rezaeian Farashahi, Ed Geraghty, Florian Hess, Nick Howgrave-Graham, Ellen Jochemsz, Thomas Johansson, Georgios Kafanas, Parimal Kumar, Jake Longo Galea, Eugene Luks, Vadim Lyubashevsky, David McCann, Bruce McIntosh, John Malone-Lee, Wenbo Mao, Dan Martin, John Merriman, Phong Nguyen, Emmanuela Orsini, Dan Page, Christopher Peikert, Joop van de Pol, David Rankin, Vincent Rijmen, Ron Rivest, Michal Rybar, Berry Schoenmakers, Tom Shrimpton, Martijn Stam, Ryan Stanley, Damien Stehle, Edlyn Teske, Susan Thomson, Frederik Vercauteren, Bogdan Warinschi, Carolyn Whitnall, Steve Williams and Marcin Wójcik.

Nigel Smart
University of Bristol

Further Reading

After finishing this book if you want to know more technical details then I would suggest the following books:

A.J. Menezes, P. van Oorschot and S.A. Vanstone. *The Handbook of Applied Cryptography.* CRC Press, 1997.

J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols.* CRC Press, 2007.

Contents

Preface v

Part 1. Mathematical Background 1

Chapter 1. Modular Arithmetic, Groups, Finite Fields and Probability 3
1.1. Modular Arithmetic 3
1.2. Finite Fields 8
1.3. Basic Algorithms 11
1.4. Probability 21
1.5. Big Numbers 24

Chapter 2. Primality Testing and Factoring 27
2.1. Prime Numbers 27
2.2. The Factoring and Factoring-Related Problems 32
2.3. Basic Factoring Algorithms 38
2.4. Modern Factoring Algorithms 42
2.5. Number Field Sieve 44

Chapter 3. Discrete Logarithms 51
3.1. The DLP, DHP and DDH Problems 51
3.2. Pohlig–Hellman 54
3.3. Baby-Step/Giant-Step Method 57
3.4. Pollard-Type Methods 59
3.5. Sub-exponential Methods for Finite Fields 64

Chapter 4. Elliptic Curves 67
4.1. Introduction 67
4.2. The Group Law 69
4.3. Elliptic Curves over Finite Fields 72
4.4. Projective Coordinates 74
4.5. Point Compression 75
4.6. Choosing an Elliptic Curve 77

Chapter 5. Lattices 79
5.1. Lattices and Lattice Reduction 79
5.2. "Hard" Lattice Problems 85
5.3. q-ary Lattices 89
5.4. Coppersmith's Theorem 90

Chapter 6. Implementation Issues 95
6.1. Introduction 95
6.2. Exponentiation Algorithms 95
6.3. Special Exponentiation Methods 99

6.4. Multi-precision Arithmetic 101
6.5. Finite Field Arithmetic 107

Part 2. Historical Ciphers 117

Chapter 7. Historical Ciphers 119
7.1. Introduction 119
7.2. Shift Cipher 120
7.3. Substitution Cipher 123
7.4. Vigenère Cipher 126
7.5. A Permutation Cipher 131

Chapter 8. The Enigma Machine 133
8.1. Introduction 133
8.2. An Equation for the Enigma 136
8.3. Determining the Plugboard Given the Rotor Settings 137
8.4. Double Encryption of Message Keys 140
8.5. Determining the Internal Rotor Wirings 141
8.6. Determining the Day Settings 147
8.7. The Germans Make It Harder 148
8.8. Known Plaintext Attack and the Bombes 150
8.9. Ciphertext Only Attack 158

Chapter 9. Information-Theoretic Security 163
9.1. Introduction 163
9.2. Probability and Ciphers 164
9.3. Entropy 169
9.4. Spurious Keys and Unicity Distance 173

Chapter 10. Historical Stream Ciphers 179
10.1. Introduction to Symmetric Ciphers 179
10.2. Stream Cipher Basics 181
10.3. The Lorenz Cipher 182
10.4. Breaking the Lorenz Cipher's Wheels 188
10.5. Breaking a Lorenz Cipher Message 192

Part 3. Modern Cryptography Basics 195

Chapter 11. Defining Security 197
11.1. Introduction 197
11.2. Pseudo-random Functions and Permutations 197
11.3. One-Way Functions and Trapdoor One-Way Functions 201
11.4. Public Key Cryptography 202
11.5. Security of Encryption 203
11.6. Other Notions of Security 209
11.7. Authentication: Security of Signatures and MACs 215
11.8. Bit Security 219
11.9. Computational Models: The Random Oracle Model 221

Chapter 12. Modern Stream Ciphers 225
12.1. Stream Ciphers from Pseudo-random Functions 225
12.2. Linear Feedback Shift Registers 227

12.3. Combining LFSRs 233
12.4. RC4 238

Chapter 13. Block Ciphers and Modes of Operation 241
13.1. Introduction to Block Ciphers 241
13.2. Feistel Ciphers and DES 244
13.3. AES 250
13.4. Modes of Operation 254
13.5. Obtaining Chosen Ciphertext Security 266

Chapter 14. Hash Functions, Message Authentication Codes and Key Derivation Functions 271
14.1. Collision Resistance 271
14.2. Padding 275
14.3. The Merkle–Damgård Construction 276
14.4. The MD-4 Family 278
14.5. HMAC 282
14.6. Merkle–Damgård-Based Key Derivation Function 284
14.7. MACs and KDFs Based on Block Ciphers 285
14.8. The Sponge Construction and SHA-3 288

Chapter 15. The "Naive" RSA Algorithm 295
15.1. "Naive" RSA Encryption 295
15.2. "Naive" RSA Signatures 299
15.3. The Security of RSA 301
15.4. Some Lattice-Based Attacks on RSA 305
15.5. Partial Key Exposure Attacks on RSA 309
15.6. Fault Analysis 310

Chapter 16. Public Key Encryption and Signature Algorithms 313
16.1. Passively Secure Public Key Encryption Schemes 313
16.2. Random Oracle Model, OAEP and the Fujisaki–Okamoto Transform 319
16.3. Hybrid Ciphers 324
16.4. Constructing KEMs 329
16.5. Secure Digital Signatures 333
16.6. Schemes Avoiding Random Oracles 342

Chapter 17. Cryptography Based on Really Hard Problems 349
17.1. Cryptography and Complexity Theory 349
17.2. Knapsack-Based Cryptosystems 353
17.3. Worst-Case to Average-Case Reductions 356
17.4. Learning With Errors (LWE) 360

Chapter 18. Certificates, Key Transport and Key Agreement 369
18.1. Introduction 369
18.2. Certificates and Certificate Authorities 371
18.3. Fresh Ephemeral Symmetric Keys from Static Symmetric Keys 375
18.4. Fresh Ephemeral Symmetric Keys from Static Public Keys 382
18.5. The Symbolic Method of Protocol Analysis 388
18.6. The Game-Based Method of Protocol Analysis 392

Part 4. Advanced Protocols 401

Chapter 19. Secret Sharing Schemes 403

19.1. Access Structures 403
19.2. General Secret Sharing 405
19.3. Reed–Solomon Codes 407
19.4. Shamir Secret Sharing 412
19.5. Application: Shared RSA Signature Generation 414

Chapter 20. Commitments and Oblivious Transfer 417
20.1. Introduction 417
20.2. Commitment Schemes 417
20.3. Oblivious Transfer 421

Chapter 21. Zero-Knowledge Proofs 425
21.1. Showing a Graph Isomorphism in Zero-Knowledge 425
21.2. Zero-Knowledge and $\mathcal{NP}$ 428
21.3. Sigma Protocols 429
21.4. An Electronic Voting System 436

Chapter 22. Secure Multi-party Computation 439
22.1. Introduction 439
22.2. The Two-Party Case 441
22.3. The Multi-party Case: Honest-but-Curious Adversaries 445
22.4. The Multi-party Case: Malicious Adversaries 448

Appendix 451

Basic Mathematical Terminology 453
A.1. Sets 453
A.2. Relations 453
A.3. Functions 455
A.4. Permutations 456
A.5. Operations 459
A.6. Groups 461
A.7. Rings 468
A.8. Fields 469
A.9. Vector Spaces 470

Index 475

Part 1

Mathematical Background

Before we tackle cryptography we need to cover some basic facts from mathematics. Much of the following can be found in a number of university "Discrete Mathematics" courses aimed at Computer Science or Engineering students, hence one hopes not all of this section is new. This part is mainly a quick overview to allow you to start on the main contents, hence you may want to first start on Part 2 and return to Part 1 when you meet some concept you are not familiar with. However, I would suggest reading Section 2.2 of Chapter 2 and Section 3.1 of Chapter 3 at least, before passing on to the rest of the book. For those who want more formal definitions of concepts, there is the appendix at the end of the book.

CHAPTER 1

Modular Arithmetic, Groups, Finite Fields and Probability

Chapter Goals

- To understand modular arithmetic.
- To become acquainted with groups and finite fields.
- To learn about basic techniques such as Euclid's algorithm, the Chinese Remainder Theorem and Legendre symbols.
- To recap basic ideas from probability theory.

1.1. Modular Arithmetic

Much of this book will be spent looking at the applications of modular arithmetic, since it is fundamental to modern cryptography and public key cryptosystems in particular. Hence, in this chapter we introduce the basic concepts and techniques we shall require.

The idea of modular arithmetic is essentially very simple and is identical to the "clock arithmetic" you learn in school. For example, converting between the 24-hour and the 12-hour clock systems is easy. One takes the value in the 24-hour clock system and reduces the hour by 12. For example 13:00 in the 24-hour clock system is one o'clock in the 12-hour clock system, since 13 modulo 12 is equal to one.

More formally, we fix a positive integer N which we call the *modulus*. For two integers a and b we write $a = b \pmod N$ if N divides $b - a$, and we say that a and b are *congruent* modulo N. Often we are lazy and just write $a = b$, if it is clear we are working modulo N.

We can also consider `(mod N)` as a postfix operator on an integer which returns the smallest non-negative value equal to the argument modulo N. For example

$$\begin{aligned} 18 \pmod 7 &= 4, \\ -18 \pmod 7 &= 3. \end{aligned}$$

The modulo operator is like the C operator %, except that in this book we usually take representatives which are non-negative. For example in C or Java we have,

```
(-3)%2 = -1
```

whilst we shall assume that $(-3) \pmod 2 = 1$.

For convenience we define the set

$$\mathbb{Z}/N\mathbb{Z} = \{0, \ldots, N-1\}$$

as the set of remainders modulo N. This is the set of values produced by the postfix operator `(mod N)`. Note, some authors use the alternative notation of $\mathbb{Z}_N$ for the set $\mathbb{Z}/N\mathbb{Z}$, however, in this book we shall stick to $\mathbb{Z}/N\mathbb{Z}$. For any set S we let $\#S$ denote the number of elements in the set S, thus $\#(\mathbb{Z}/N\mathbb{Z}) = N$.

The set $\mathbb{Z}/N\mathbb{Z}$ has two basic operations on it, namely addition and multiplication. These are defined in the obvious way, for example:

$$(11 + 13) \pmod{16} = 24 \pmod{16} = 8$$

since $24 = 1 \cdot 16 + 8$ and

$$(11 \cdot 13) \pmod{16} = 143 \pmod{16} = 15$$

since $143 = 8 \cdot 16 + 15$.

1.1.1. Groups: Addition and multiplication modulo N work almost the same as arithmetic over the reals or the integers. In particular we have the following properties:

(1) Addition is closed:
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a + b \in \mathbb{Z}/N\mathbb{Z}.$$
(2) Addition is associative:
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a + b) + c = a + (b + c).$$
(3) 0 is an additive identity:
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a + 0 = 0 + a = a.$$
(4) The additive inverse always exists:
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a + (N - a) = (N - a) + a = 0,$$
i.e. $-a$ is an element which when combined with a produces the additive identity.
(5) Addition is commutative:
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a + b = b + a.$$
(6) Multiplication is closed:
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot b \in \mathbb{Z}/N\mathbb{Z}.$$
(7) Multiplication is associative:
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a \cdot b) \cdot c = a \cdot (b \cdot c).$$
(8) 1 is a multiplicative identity:
$$\forall a \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot 1 = 1 \cdot a = a.$$
(9) Multiplication and addition satisfy the distributive law:
$$\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : \ (a + b) \cdot c = a \cdot c + b \cdot c.$$
(10) Multiplication is commutative:
$$\forall a, b \in \mathbb{Z}/N\mathbb{Z} : \ a \cdot b = b \cdot a.$$

Many of the sets we will encounter have a number of these properties, so we give special names to these sets as a shorthand.

Definition 1.1 (Groups)**.** *A group is a set with an operation on its elements which*

- *Is closed,*
- *Has an identity,*
- *Is associative, and*
- *Every element has an inverse.*

A group which is commutative is often called *abelian*. Almost all groups that one meets in cryptography are abelian, since the commutative property is often what makes them cryptographically interesting. Hence, any set with properties 1, 2, 3 and 4 above is called a group, whilst a set with properties 1, 2, 3, 4 and 5 is called an abelian group. Standard examples of groups which one meets all the time in high school are:

- The integers, the reals or the complex numbers under addition. Here the identity is 0 and the inverse of x is $-x$, since $x + (-x) = 0$.
- The non-zero rational, real or complex numbers under multiplication. Here the identity is 1 and the inverse of x is denoted by x^{-1}, since $x \cdot x^{-1} = 1$.

A group is called *multiplicative* if we tend to write its group operation in the same way as one does for multiplication, i.e.

$$f = g \cdot h \text{ and } g^5 = g \cdot g \cdot g \cdot g \cdot g.$$

We use the notation $(G, \cdot)$ in this case if there is some ambiguity as to which operation on G we are considering. A group is called *additive* if we tend to write its group operation in the same way as one does for addition, i.e.

$$f = g + h \text{ and } 5 \cdot g = g + g + g + g + g.$$

In this case we use the notation $(G, +)$ if there is some ambiguity. An abelian group is called *cyclic* if there is a special element, called the *generator*, from which every other element can be obtained either by repeated application of the group operation, or by the use of the inverse operation. For example, in the integers under addition every positive integer can be obtained by repeated addition of 1 to itself, e.g. 7 can be expressed by

$$7 = 1 + 1 + 1 + 1 + 1 + 1 + 1.$$

Every negative integer can be obtained from a positive integer by application of the additive inverse operator, which sends x to $-x$. Hence, we have that 1 is a generator of the integers under addition.

If g is a generator of the cyclic group G we often write $G = \langle g \rangle$. If G is multiplicative then every element h of G can be written as

$$h = g^x,$$

whilst if G is additive then every element h of G can be written as

$$h = x \cdot g,$$

where x in both cases is some integer called the *discrete logarithm* of h to the base g.

1.1.2. Rings: As well as groups we also use the concept of a ring.

Definition 1.2 (Rings)**.** *A ring is a set with two operations, usually denoted by $+$ and $\cdot$ for addition and multiplication, which satisfies properties* 1 *to* 9 *above. We can denote a ring and its two operations by the triple $(R, \cdot, +)$. If it also happens that multiplication is commutative we say that the ring is commutative.*

This may seem complicated but it sums up the type of sets one deals with all the time, for example the infinite commutative rings of integers, real or complex numbers. In fact in cryptography things are even easier since we only need to consider finite rings, like the commutative ring of integers modulo N, $\mathbb{Z}/N\mathbb{Z}$. Thus $\mathbb{Z}/N\mathbb{Z}$ is an abelian group when we only think of addition, but it is also a ring if we want to worry about multiplication as well.

1.1.3. Euler's ϕ Function: In modular arithmetic it will be important to know when, given a and b, the equation

$$a \cdot x = b \pmod{N}$$

has a solution. For example there is exactly one solution in the set $\mathbb{Z}/143\mathbb{Z} = \{0, \ldots, 142\}$ to the equation

$$7 \cdot x = 3 \pmod{143},$$

but there are no solutions to the equation

$$11 \cdot x = 3 \pmod{143},$$

however there are 11 solutions to the equation

$$11 \cdot x = 22 \pmod{143}.$$

Luckily, it is very easy to test when such an equation has one, many or no solutions. We simply compute the greatest common divisor, or gcd, of a and N, i.e. $\gcd(a, N)$.

- If $\gcd(a, N) = 1$ then there is exactly one solution. We find the value c such that $a \cdot c = 1 \pmod{N}$ and then we compute $x \leftarrow b \cdot c \pmod{N}$.
- If $g = \gcd(a, N) \neq 1$ and $\gcd(a, N)$ divides b then there are g solutions. Here we divide the whole equation by g to produce the equation

  $$a' \cdot x' = b' \pmod{N'},$$

  where $a' = a/g$, $b' = b/g$ and $N' = N/g$. If x' is a solution to the above equation then

  $$x \leftarrow x' + i \cdot N'$$

  for $0 \leq i < g$ is a solution to the original one.
- Otherwise there are no solutions.

The case where $\gcd(a, N) = 1$ is so important we have a special name for it: we say a and N are relatively prime or coprime.

In the above description we wrote $x \leftarrow y$ to mean that we *assign* x the value y; this is to distinguish it from saying $x = y$, by which we mean x and y are equal. Clearly after assignment of y to x the values of x and y are indeed equal. But imagine we wanted to increment x by one, we would write $x \leftarrow x + 1$, the meaning of which is clear. Whereas $x = x + 1$ is possibly a statement which evaluates to false!

Another reason for this special notation for assignment is that we can extend it to algorithms, or procedures. So for example $x \leftarrow A(z)$ might mean we assign x the output of procedure A on input of z. This procedure might be randomized, and in such a case we are thereby assuming an implicit probability distribution of the output x. We might even write $x \leftarrow S$ where S is some set, by which we mean we assign x a value from the set S chosen uniformly at random. Thus our original $x \leftarrow y$ notation is just a shorthand for $x \leftarrow \{y\}$.

The number of integers in $\mathbb{Z}/N\mathbb{Z}$ which are relatively prime to N is given by the Euler ϕ function, $\phi(N)$. Given the prime factorization of N it is easy to compute the value of $\phi(N)$. If N has the prime factorization

$$N = \prod_{i=1}^{n} p_i^{e_i}$$

then

$$\phi(N) = \prod_{i=1}^{n} p_i^{e_i - 1}(p_i - 1).$$

Note, the last statement is very important for cryptography: *Given the factorization of N it is easy to compute the value of $\phi(N)$*. The most important cases for the value of $\phi(N)$ in cryptography are:

(1) If p is prime then

$$\phi(p) = p - 1.$$

(2) If p and q are both prime and $p \neq q$ then

$$\phi(p \cdot q) = (p - 1)(q - 1).$$

1.1.4. Multiplicative Inverse Modulo N: We have just seen that when we wish to solve equations of the form

$$a \cdot x = b \pmod{N}$$

we reduce the problem to the question of examining whether a has a multiplicative inverse modulo N, i.e. whether there is a number c such that

$$a \cdot c = c \cdot a = 1 \pmod{N}.$$

Such a value of c is often written a^{-1}. Clearly a^{-1} is the solution to the equation

$$a \cdot x = 1 \pmod{N}.$$

Hence, the inverse of a only exists when a and N are coprime, i.e. $\gcd(a, N) = 1$. Of particular interest is when N is a prime p, since then for all non-zero values of $a \in \mathbb{Z}/p\mathbb{Z}$ we always obtain a unique solution to

$$a \cdot x = 1 \pmod{p}.$$

Hence, if p is a prime then every non-zero element in $\mathbb{Z}/p\mathbb{Z}$ has a multiplicative inverse. A ring like $\mathbb{Z}/p\mathbb{Z}$ with this property is called a field.

Definition 1.3 (Fields)**.** *A field is a set with two operations $(G, \cdot, +)$ such that*

- *$(G, +)$ is an abelian group with identity denoted by 0,*
- *$(G \setminus \{0\}, \cdot)$ is an abelian group,*
- *$(G, \cdot, +)$ satisfies the distributive law.*

Hence, a field is a commutative ring for which every non-zero element has a multiplicative inverse. You have met fields before, for example consider the infinite fields of rational, real or complex numbers.

1.1.5. The Set $(\mathbb{Z}/N\mathbb{Z})^*$: We define the set of all invertible elements in $\mathbb{Z}/N\mathbb{Z}$ by

$$(\mathbb{Z}/N\mathbb{Z})^* = \{x \in \mathbb{Z}/N\mathbb{Z} : \gcd(x, N) = 1\}.$$

The * in A^*, for any ring A, refers to the largest subset of A which forms a group under multiplication. Hence, the set $(\mathbb{Z}/N\mathbb{Z})^*$ is a group with respect to multiplication and it has size $\phi(N)$. In the special case when N is a prime p we have

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1, \ldots, p-1\}$$

since every non-zero element of $\mathbb{Z}/p\mathbb{Z}$ is coprime to p. For an arbitrary field F the set F^* is equal to the set $F \setminus \{0\}$. To ease notation, for this very important case, we define

$$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\}$$

and

$$\mathbb{F}_p^* = (\mathbb{Z}/p\mathbb{Z})^* = \{1, \ldots, p-1\}.$$

The set $\mathbb{F}_p$ is said to be a finite field of characteristic p. In the next section we shall discuss a more general type of finite field, but for now recall the important point that the integers modulo N are only a field when N is a prime. We end this section with the most important theorem in elementary group theory.

Theorem 1.4 (Lagrange's Theorem)**.** *If $(G, \cdot)$ is a group of order (size) $n = \#G$ then for all $a \in G$ we have $a^n = 1$.*

So if $x \in (\mathbb{Z}/N\mathbb{Z})^*$ then

$$x^{\phi(N)} = 1 \pmod{N}$$

since $\#(\mathbb{Z}/N\mathbb{Z})^* = \phi(N)$. This leads us to Fermat's Little Theorem, not to be confused with Fermat's Last Theorem which is something entirely different.

Theorem 1.5 (Fermat's Little Theorem)**.** *Suppose p is a prime and $a \in \mathbb{Z}$, then*

$$a^p = a \pmod{p}.$$

Fermat's Little Theorem is a special case of Lagrange's Theorem and will form the basis of one of the primality tests considered in a later chapter.

1.2. Finite Fields

The integers modulo a prime p are not the only type of finite field. In this section we shall introduce another type of finite field which is particularly important. At first reading you may wish to skip this section. We shall only be using these general forms of finite fields when discussing the AES block cipher, stream ciphers based on linear feedback shift registers and when we look at systems based on elliptic curves.

For this section we let p denote a prime number. Consider the set of polynomials in X whose coefficients are elements of $\mathbb{F}_p$. We denote this set $\mathbb{F}_p[X]$, which forms a ring with the natural definition of addition and multiplication of polynomials modulo p. Of particular interest is the case when $p = 2$, from which we draw most of our examples in this section. For example, in $\mathbb{F}_2[X]$ we have

$$\begin{aligned}(1 + X + X^2) + (X + X^3) &= 1 + X^2 + X^3,\\ (1 + X + X^2) \cdot (X + X^3) &= X + X^2 + X^4 + X^5.\end{aligned}$$

Just as with the integers modulo a number N, where the integers modulo N formed a ring, we can take a polynomial $f(X)$ and then the polynomials modulo $f(X)$ also form a ring. We denote this ring by

$$\mathbb{F}_p[X]/f(X)\mathbb{F}_p[X]$$

or more simply

$$\mathbb{F}_p[X]/(f(X)).$$

But to ease notation we will often write $\mathbb{F}_p[X]/f(X)$ for this latter ring. When $f(X) = X^4 + 1$ and $p = 2$ we have, for example,

$$(1 + X + X^2) \cdot (X + X^3) \pmod{X^4 + 1} = 1 + X^2$$

since

$$X + X^2 + X^4 + X^5 = (X + 1) \cdot (X^4 + 1) + (1 + X^2).$$

When checking the above equation you should remember we are working modulo two.

1.2.1. Inversion in General Finite Fields: Recall, when we looked at the integers modulo N we looked at the equation $a \cdot x = b \pmod{N}$. We can consider a similar question for polynomials. Given a, b and f, all of which are polynomials in $\mathbb{F}_p[X]$, does there exist a solution α to the equation $a \cdot \alpha = b \pmod{f}$? With integers the answer depended on the greatest common divisor of a and f, and we counted three possible cases. A similar three cases can occur for polynomials, with the most important one being when a and f are coprime and so have greatest common divisor equal to one.

A polynomial is called irreducible if it has no proper factors other than itself and the constant polynomials. Hence, irreducibility of polynomials is the same as primality of numbers. Just as with the integers modulo N, when N was prime we obtained a finite field, so when $f(X)$ is irreducible the ring $\mathbb{F}_p[X]/f(X)$ also forms a finite field.

1.2.2. Isomorphisms of Finite Fields: Consider the case $p = 2$ and the two different irreducible polynomials

$$f_1 = X^7 + X + 1$$

and

$$f_2 = Y^7 + Y^3 + 1.$$

Now, consider the two finite fields

$$F_1 = \mathbb{F}_2[X]/f_1(X) \text{ and } F_2 = \mathbb{F}_2[Y]/f_2(Y).$$

These both consist of the 2^7 binary polynomials of degree less than seven. Addition in these two fields is identical in that one just adds the coefficients of the polynomials modulo two. The only difference is in how multiplication is performed

$$\begin{aligned}(X^3 + 1) \cdot (X^4 + 1) \pmod{f_1(X)} &= X^4 + X^3 + X,\\ (Y^3 + 1) \cdot (Y^4 + 1) \pmod{f_2(Y)} &= Y^4.\end{aligned}$$

A natural question arises as to whether these fields are "really" different, or whether they just "look" different. In mathematical terms the question is whether the two fields are *isomorphic*. It turns out that they are isomorphic if there is a map

$$\phi : F_1 \longrightarrow F_2,$$

called a field isomorphism, which satisfies

$$\begin{aligned}\phi(\alpha + \beta) &= \phi(\alpha) + \phi(\beta),\\ \phi(\alpha \cdot \beta) &= \phi(\alpha) \cdot \phi(\beta).\end{aligned}$$

Such an isomorphism exists for every two finite fields of the same order, although we will not show it here. To describe the map above you only need to show how to express a root of $f_2(Y)$ in terms of a polynomial in the root of $f_1(X)$, with the inverse map being a polynomial which expresses a root of $f_1(X)$ in terms of a polynomial in the root of $f_2(Y)$, i.e.

$$\begin{aligned}Y = g_1(X) &= X + X^2 + X^3 + X^5,\\ X = g_2(Y) &= Y^5 + Y^4.\end{aligned}$$

Notice that $g_2(g_1(X)) \pmod{f_1(X)} = X$, that $f_2(g_1(X)) \pmod{f_1(X)} = 0$ and that $f_1(g_2(Y)) \pmod{f_2(Y)} = 0$.

One can show that all finite fields of the same characteristic and prime are isomorphic, thus we have the following.

Theorem 1.6. *There is (up to isomorphism) just one finite field of each prime power order.*

The notation we use for these fields is either $\mathbb{F}_q$ or $GF(q)$, with $q = p^d$ where d is the degree of the irreducible polynomial used to construct the field; we of course have $\mathbb{F}_p = \mathbb{F}_p[X]/X$. The notation $GF(q)$ means the Galois field of q elements, in honour of the nineteenth century French mathematician Galois. Galois had an interesting life; he accomplished his scientific work at an early age before dying in a duel.

1.2.3. Field Towers and the Frobenius Map: There are a number of technical definitions associated with finite fields which we need to cover. A subset F of a field K is called a *subfield* if F is a field with respect to the same operations for which K is a field. Each finite field K contains a copy of the integers modulo p for some prime p, i.e. $\mathbb{F}_p \subset K$. We call this prime the *characteristic* of the field, and often write this as char K. The subfield of integers modulo p of a finite field is called the prime subfield.

There is a map Φ called the pth power *Frobenius map* defined for any finite field by

$$\Phi : \begin{cases} \mathbb{F}_q \longrightarrow \mathbb{F}_q \\ \alpha \longmapsto \alpha^p \end{cases}$$

where p is the characteristic of $\mathbb{F}_q$. The Frobenius map is an isomorphism of $\mathbb{F}_q$ with itself; such an isomorphism is called an automorphism. An interesting property is that the set of elements fixed by the Frobenius map is the prime field, i.e.

$$\{\alpha \in \mathbb{F}_q : \alpha^p = \alpha\} = \mathbb{F}_p.$$

Notice that this is a kind of generalization of Fermat's Little Theorem to finite fields. For any automorphism χ of a finite field, the set of elements fixed by χ is a field, called the fixed field of χ. Hence the previous statement says that the fixed field of the Frobenius map is the prime field $\mathbb{F}_p$.

Not only does $\mathbb{F}_q$ contain a copy of $\mathbb{F}_p$ but $\mathbb{F}_{p^d}$ contains a copy of $\mathbb{F}_{p^e}$ for every value of e dividing d; see Figure 1.1 for an example. In addition $\mathbb{F}_{p^e}$ is the fixed field of the automorphism Φ^e, i.e.

$$\{\alpha \in \mathbb{F}_{p^d} : \alpha^{p^e} = \alpha\} = \mathbb{F}_{p^e}.$$

If we define $\mathbb{F}_q$ as $\mathbb{F}_p[X]/f(X)$, for some irreducible polynomial $f(X)$ with $p^{\deg f} = q$, then another

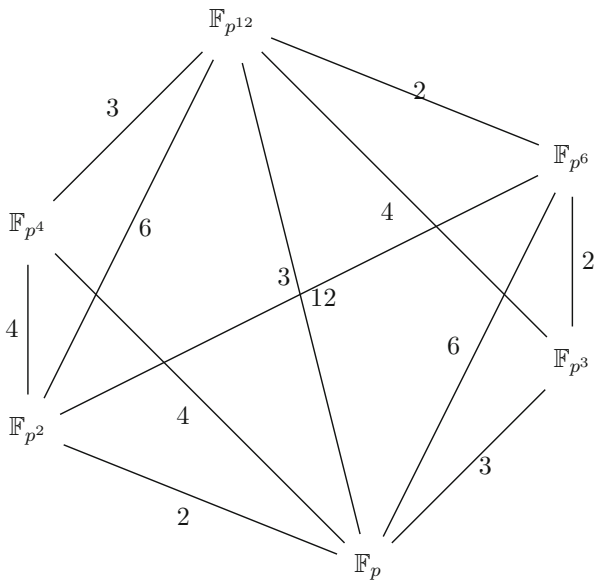


FIGURE 1.1. Example tower of finite fields. The number on each line gives the degree of the subfield within the larger field

way of thinking of $\mathbb{F}_q$ is as the set of polynomials of degree less than $\deg f$ in a root of $f(X)$. In other words let α be a "formal" root of $f(X)$, then we define

$$\mathbb{F}_q = \{ \sum_{i=0}^{\deg f - 1} a_i \cdot \alpha^i : a_i \in \mathbb{F}_p \}$$

with addition being addition of polynomials modulo p, and multiplication being polynomial multiplication modulo p, subject to the fact that $f(\alpha) = 0$. To see why this amounts to the same object

take two polynomials $a(X)$ and $b(X)$ and let $c(X) = a(X) \cdot b(X) \pmod{f(X)}$. Then there is a polynomial $q(X)$ such that

$$c(X) = a(X) \cdot b(X) + q(X) \cdot f(X),$$

which is our multiplication method given in terms of polynomials. In terms of a root α of $f(X)$ we note that we have

$$\begin{aligned} c(\alpha) &= a(\alpha) \cdot b(\alpha) + q(\alpha) \cdot f(\alpha). \\ &= a(\alpha) \cdot b(\alpha) + q(\alpha) \cdot 0, \\ &= a(\alpha) \cdot b(\alpha). \end{aligned}$$

Another interesting property is that if p is the characteristic of $\mathbb{F}_q$ then if we take any element $\alpha \in \mathbb{F}_q$ and add it to itself p times we obtain zero, e.g. in $\mathbb{F}_{49}$ we have

$$X + X + X + X + X + X + X = 7 \cdot X = 0 \pmod{7}.$$

The non-zero elements of a finite field, usually denoted $\mathbb{F}_q^*$, form a cyclic finite abelian group, called the multiplicative group of the finite field. We call a generator of $\mathbb{F}_q^*$ a primitive element in the finite field. Such primitive elements always exist, and indeed there are $\phi(q)$ of them, and so the multiplicative group is always cyclic. In other words there always exists an element $g \in \mathbb{F}_q$ such that every non-zero element α can be written as

$$\alpha = g^x$$

for some integer value of x.

Example: As an example consider the field of eight elements defined by

$$\mathbb{F}_{2^3} = \mathbb{F}_2[X]/(X^3 + X + 1).$$

In this field there are seven non-zero elements; namely

$$1, \alpha, \alpha + 1, \alpha^2, \alpha^2 + 1, \alpha^2 + \alpha, \alpha^2 + \alpha + 1$$

where α is a root of $X^3 + X + 1$. We see that α is a primitive element in $\mathbb{F}_{2^3}$ since

$$\begin{aligned} \alpha^1 &= \alpha, \\ \alpha^2 &= \alpha^2, \\ \alpha^3 &= \alpha + 1, \\ \alpha^4 &= \alpha^2 + \alpha, \\ \alpha^5 &= \alpha^2 + \alpha + 1, \\ \alpha^6 &= \alpha^2 + 1, \\ \alpha^7 &= 1. \end{aligned}$$

Notice that for a prime p this means that the integers modulo p also have a primitive element, since $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$ is a finite field.

1.3. Basic Algorithms

There are several basic numerical algorithms or techniques which everyone should know since they occur in many places in this book. The ones we shall concentrate on here are

- Euclid's gcd algorithm,
- The Chinese Remainder Theorem,
- Computing Jacobi and Legendre symbols.

1.3.1. Greatest Common Divisors: In the previous sections we said that when trying to solve

$$a \cdot x = b \pmod{N}$$

in integers, or

$$a \cdot \alpha = b \pmod{f}$$

for polynomials modulo a prime, we needed to compute the greatest common divisor. This was particularly important in determining whether $a \in \mathbb{Z}/N\mathbb{Z}$ or $a \in \mathbb{F}_p[X]/f$ had a multiplicative inverse or not, i.e. $\gcd(a, N) = 1$ or $\gcd(a, f) = 1$. We did not explain how this greatest common divisor is computed, neither did we explain how the inverse is to be computed when we know it exists. We shall now address this omission by explaining one of the oldest algorithms known to man, namely the Euclidean algorithm.

If we were able to factor a and N into primes, or a and f into irreducible polynomials, then computing the greatest common divisor would be particularly easy. For example if we were given

$$\begin{aligned} a &= 230\,895\,588\,646\,864 = 2^4 \cdot 157 \cdot 4513^3, \\ b &= 33\,107\,658\,350\,407\,876 = 2^2 \cdot 157 \cdot 2269^3 \cdot 4513, \end{aligned}$$

then it is easy, from the factorization, to compute the gcd as

$$\gcd(a, b) = 2^2 \cdot 157 \cdot 4513 = 2\,834\,164.$$

However, factoring is an expensive operation for integers, so the above method is very slow for large integers. However, computing greatest common divisors is actually easy as we shall now show. Although factoring for polynomials modulo a prime is very easy, it turns out that almost all algorithms to factor polynomials require access to an algorithm to compute greatest common divisors. Hence, in both situations we need to be able to compute greatest common divisors without recourse to factoring.

1.3.2. The Euclidean Algorithm: In the following we will consider the case of integers only; the generalization to polynomials is easy since both integers and polynomials allow Euclidean division. For integers a and b, Euclidean division is the operation of finding q and r with $0 \leq r < |b|$ such that

$$a = q \cdot b + r,$$

i.e. $r \leftarrow a \pmod{b}$. For polynomials f and g, Euclidean division means finding polynomials q, r with $0 \leq \deg r < \deg g$ such that

$$f = q \cdot g + r.$$

To compute the gcd of $r_0 = a$ and $r_1 = b$ we compute $r_2, r_3, r_4, \ldots$ by $r_{i+2} = r_i \pmod{r_{i+1}}$, until $r_{m+1} = 0$, so we have:

$$\begin{aligned} r_2 &\leftarrow r_0 - q_1 \cdot r_1, \\ r_3 &\leftarrow r_1 - q_2 \cdot r_2, \\ &\vdots \quad \vdots \\ r_m &\leftarrow r_{m-2} - q_{m-1} \cdot r_{m-1}, \\ r_{m+1} &\leftarrow 0, \qquad\qquad \text{i.e. } r_m \text{ divides } r_{m-1}. \end{aligned}$$

If d divides a and b then d divides r_2, r_3, r_4 and so on. Hence

$$\gcd(a, b) = \gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_{m-1}, r_m) = r_m.$$

As an example of this algorithm we can show that $3 = \gcd(21, 12)$. Using the Euclidean algorithm we compute $\gcd(21, 12)$ in the steps

$$\begin{aligned}\gcd(21, 12) &= \gcd(21 \pmod{12}, 12)\\ &= \gcd(9, 12)\\ &= \gcd(12 \pmod{9}, 9)\\ &= \gcd(3, 9)\\ &= \gcd(9 \pmod{3}, 3)\\ &= \gcd(0, 3) = 3.\end{aligned}$$

Or, as an example with larger numbers,

$$\begin{aligned}\gcd(1\,426\,668\,559\,730,\ 810\,653\,094\,756) &= \gcd(810\,653\,094\,756,\ 616\,015\,464\,974),\\ &= \gcd(616\,015\,464\,974,\ 194\,637\,629\,782),\\ &= \gcd(194\,637\,629\,782,\ 32\,102\,575\,628),\\ &= \gcd(32\,102\,575\,628,\ 2\,022\,176\,014),\\ &= \gcd(2\,022\,176\,014,\ 1\,769\,935\,418),\\ &= \gcd(1\,769\,935\,418,\ 252\,240\,596),\\ &= \gcd(252\,240\,596,\ 4\,251\,246),\\ &= \gcd(4\,251\,246,\ 1\,417\,082),\\ &= \gcd(1\,417\,082,\ 0),\\ &= 1\,417\,082.\end{aligned}$$

The Euclidean algorithm essentially works because the mapping

$$(a, b) \longmapsto (a \pmod{b}, b),$$

for $a \geq b$ is a gcd-preserving mapping, i.e. the input and output of pairs of integers from the mapping have the same greatest common divisor. In computer science terms the greatest common divisor is an invariant of the mapping. In addition for inputs $a, b > 0$ the algorithm terminates since the mapping produces a sequence of decreasing non-negative integers, which must eventually end up with the smallest value being zero.

The trouble with the above method for determining a greatest common divisor is that computers find it much easier to add and multiply numbers than to take remainders or quotients. Hence, implementing a gcd algorithm with the above gcd-preserving mapping will usually be very inefficient. Fortunately, there are a number of other gcd-preserving mappings: For example the following is a gcd-preserving mapping between pairs of integers, which are not both even,

$$(a, b) \longmapsto \begin{cases}((a-b)/2, b) & \text{If } a \text{ and } b \text{ are odd.}\\ (a/2, b) & \text{If } a \text{ is even and } b \text{ is odd.}\\ (a, b/2) & \text{If } a \text{ is odd and } b \text{ is even.}\end{cases}$$

Recall that computers find it easy to divide by two, since in binary this is accomplished by a cheap bit shift operation. This latter mapping gives rise to the binary Euclidean algorithm, which is the one usually implemented on a computer. Essentially, this algorithm uses the above gcd-preserving mapping after first removing any power of two in the gcd. Algorithm 1.1 explains how this works, on input of two positive integers a and b.

Algorithm 1.1: Binary Euclidean algorithm

```
g ← 1.
/* Remove powers of two from the gcd */
while (a mod 2 = 0) and (b mod 2 = 0) do
    a ← a/2, b ← b/2, g ← 2 · g.
/* At least one of a and b is now odd */
while a ≠ 0 do
    while a mod 2 = 0 do a ← a/2.
    while b mod 2 = 0 do b ← b/2.
    /* Now both a and b are odd */
    if a ≥ b then a ← (a − b)/2.
    else b ← (b − a)/2.
return g · b
```

1.3.3. The Extended Euclidean Algorithm: Using the Euclidean algorithm we can determine when a has an inverse modulo N by testing whether

$$\gcd(a, N) = 1.$$

But we still do not know how to determine the inverse when it exists. To do this we use a variant of Euclid's gcd algorithm, called the extended Euclidean algorithm. Recall we had

$$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$$

with $r_m = \gcd(r_0, r_1)$. Now we unwind the above and write each r_i, for $i \geq 2$, in terms of a and b. So we have the identities

$$\begin{aligned}
r_2 &= r_0 - q_1 \cdot r_1 = a - q_1 \cdot b \\
r_3 &= r_1 - q_2 \cdot r_2 = b - q_2 \cdot (a - q_1 \cdot b) = -q_2 \cdot a + (1 + q_1 \cdot q_2) \cdot b \\
&\vdots \quad \vdots \\
r_{i-2} &= s_{i-2} \cdot a + t_{i-2} \cdot b \\
r_{i-1} &= s_{i-1} \cdot a + t_{i-1} \cdot b \\
r_i &= r_{i-2} - q_{i-1} \cdot r_{i-1} \\
&= a \cdot (s_{i-2} - q_{i-1} \cdot s_{i-1}) + b \cdot (t_{i-2} - q_{i-1} \cdot t_{i-1}) \\
&\vdots \quad \vdots \\
r_m &= s_m \cdot a + t_m \cdot b.
\end{aligned}$$

The extended Euclidean algorithm takes as input a and b and outputs values r_m, s_m and t_m such that

$$r_m = \gcd(a, b) = s_m \cdot a + t_m \cdot b.$$

Hence, we can now solve our original problem of determining the inverse of a modulo N, when such an inverse exists. We first apply the extended Euclidean algorithm to a and $b = N$ so as to compute d, x, y such that

$$d = \gcd(a, N) = x \cdot a + y \cdot N.$$

This algorithm is described in Algorithm 1.2. The value d will be equal to one, as we have assumed that a and N are coprime. Given the output from this algorithm, we can solve the equation $a \cdot x = 1 \pmod N$, since we have $d = x \cdot a + y \cdot N = x \cdot a \pmod N$.

Algorithm 1.2: Extended Euclidean algorithm

$s \leftarrow 0, s' \leftarrow 1, t \leftarrow 1, t' \leftarrow 0, r \leftarrow b, r' \leftarrow a.$
while $r \neq 0$ **do**
  $q \leftarrow \lfloor r'/r \rfloor.$
  $(r', r) \leftarrow (r, r' - q \cdot r).$
  $(s', s) \leftarrow (s, s' - q \cdot s).$
  $(t', t) \leftarrow (t, t' - q \cdot t).$
$d \leftarrow r', x \leftarrow t, y \leftarrow s.$
return d, x, y.

As an example suppose we wish to compute the inverse of 7 modulo 19. We first set $r_0 = 7$ and $r_1 = 19$ and then we compute

$$\begin{aligned} r_2 &\leftarrow 5 = 19 - 2 \cdot 7 \\ r_3 &\leftarrow 2 = 7 - 5 = 7 - (19 - 2 \cdot 7) = -19 + 3 \cdot 7 \\ r_4 &\leftarrow 1 = 5 - 2 \cdot 2 = (19 - 2 \cdot 7) - 2 \cdot (-19 + 3 \cdot 7) = 3 \cdot 19 - 8 \cdot 7. \end{aligned}$$

Hence,

$$1 = -8 \cdot 7 \pmod{19}$$

and so

$$7^{-1} = -8 = 11 \pmod{19}.$$

Note, a binary version of the above algorithm also exists. We leave it to the reader to work out the details of the binary version of the extended Euclidean algorithm.

1.3.4. Chinese Remainder Theorem (CRT): The Chinese Remainder Theorem, or CRT, is also a very old piece of mathematics, which dates back at least 2 000 years. We shall use the CRT in a few places, for example to improve the performance of the decryption operation of RSA and in a number of other protocols. In a nutshell the CRT states that if we have the two equations

$$x = a \pmod{N} \text{ and } x = b \pmod{M}$$

then there is a unique solution modulo $(M \cdot N)$ if and only if $\gcd(N, M) = 1$. In addition it gives a method to easily find the solution. For example if the two equations are given by

$$\begin{aligned} x &= 4 \pmod{7}, \\ x &= 3 \pmod{5}, \end{aligned}$$

then we have

$$x = 18 \pmod{35}.$$

It is easy to check that this is a solution, since $18 \pmod{7} = 4$ and $18 \pmod{5} = 3$. But how did we produce this solution?

We shall first show how this can be done naively from first principles and then we shall give the general method. We have the equations

$$x = 4 \pmod{7} \text{ and } x = 3 \pmod{5}.$$

Hence for some u we have

$$x = 4 + 7 \cdot u \text{ and } x = 3 \pmod{5}.$$

Putting these latter two equations together, one obtains

$$4 + 7 \cdot u = 3 \pmod{5}.$$

We then rearrange the equation to find

$$2 \cdot u = 7 \cdot u = 3 - 4 = 4 \pmod 5.$$

Now since $\gcd(2, 5) = 1$ we can solve the above equation for u. First we compute $2^{-1} \pmod 5 = 3$, since $2 \cdot 3 = 6 = 1 \pmod 5$. Then we compute the value of $u = 2^{-1} \cdot 4 = 3 \cdot 4 = 2 \pmod 5$. Then substituting this value of u back into our equation for x gives the solution

$$x = 4 + 7 \cdot u = 4 + 7 \cdot 2 = 18.$$

The Chinese Remainder Theorem: Two Equations: The case of two equations is so important we now give a general formula. We assume that $\gcd(N, M) = 1$, and that we are given the equations

$$x = a \pmod M \text{ and } x = b \pmod N.$$

We first compute

$$T \leftarrow M^{-1} \pmod N$$

which is possible since we have assumed $\gcd(N, M) = 1$. We then compute

$$u \leftarrow (b - a) \cdot T \pmod N.$$

The solution modulo $M \cdot N$ is then given by

$$x \leftarrow a + u \cdot M.$$

To see this always works we verify

$$\begin{aligned} x \pmod M &= a + u \cdot M \pmod M \\ &= a, \\ x \pmod N &= a + u \cdot M \pmod N \\ &= a + (b - a) \cdot T \cdot M \pmod N \\ &= a + (b - a) \cdot M^{-1} \cdot M \pmod N \\ &= a + (b - a) \pmod N \\ &= b. \end{aligned}$$

The Chinese Remainder Theorem: The General Case: Now we turn to the general case of the CRT where we consider more than two equations at once. Let $m_1, \ldots, m_r$ be pairwise relatively prime and let $a_1, \ldots, a_r$ be given. We want to find x modulo $M = m_1 \cdot m_2 \cdots m_r$ such that

$$x = a_i \pmod{m_i} \text{ for all } i.$$

The Chinese Remainder Theorem guarantees a unique solution given by

$$x \leftarrow \sum_{i=1}^{r} a_i \cdot M_i \cdot y_i \pmod M$$

where

$$M_i \leftarrow M/m_i \text{ and } y_i \leftarrow M_i^{-1} \pmod{m_i}.$$

As an example suppose we wish to find the unique x modulo

$$M = 1001 = 7 \cdot 11 \cdot 13$$

such that

$$\begin{aligned} x &= 5 \pmod 7, \\ x &= 3 \pmod{11}, \\ x &= 10 \pmod{13}. \end{aligned}$$

We compute

$$\begin{aligned} M_1 &\leftarrow 143, \quad y_1 \leftarrow 5, \\ M_2 &\leftarrow 91, \quad y_2 \leftarrow 4, \\ M_3 &\leftarrow 77, \quad y_3 \leftarrow 12. \end{aligned}$$

Then, the solution is given by

$$\begin{aligned} x &\leftarrow \sum_{i=1}^{r} a_i \cdot M_i \cdot y_i \pmod{M} \\ &= 715 \cdot 5 + 364 \cdot 3 + 924 \cdot 10 \pmod{1001} \\ &= 894. \end{aligned}$$

1.3.5. The Legendre Symbol: Let p denote a prime, greater than two. Consider the mapping

$$\begin{aligned} \mathbb{F}_p &\longrightarrow \mathbb{F}_p \\ \alpha &\longmapsto \alpha^2. \end{aligned}$$

Since $-\alpha$ and α are distinct elements of $\mathbb{F}_p$ if $\alpha \neq 0$ and $p \neq 2$, and because $(-\alpha)^2 = \alpha$, we see that the mapping $\alpha \longmapsto \alpha^2$ is exactly two-to-one on the non-zero elements of $\mathbb{F}_p$. So if an element x in $\mathbb{F}_p$ has a square root, then it has exactly two square roots (unless $x = 0$) and exactly half of the elements of $\mathbb{F}_p^*$ are squares. The set of squares in $\mathbb{F}_p^*$ are called the *quadratic residues* and they form a subgroup of order $(p-1)/2$ of the multiplicative group $\mathbb{F}_p^*$. The elements of $\mathbb{F}_p^*$ which are not squares are called the *quadratic non-residues*.

To make it easy to detect squares modulo a prime p we define the *Legendre symbol*

$$\left(\frac{a}{p}\right).$$

This is defined to be equal to 0 if p divides a, equal to $+1$ if a is a quadratic residue and equal to -1 if a is a quadratic non-residue.

Notice that, if $a \neq 0$ is a square then it has order dividing $(p-1)/2$ since there is an s such that $s^2 = a$ and s has order dividing $(p-1)$ (by Lagrange's Theorem). Hence if a is a square it must have order dividing $(p-1)/2$, and so $a^{(p-1)/2} \pmod{p} = 1$. However, if a is not a square then by the same reasoning it cannot have order dividing $(p-1)/2$. We then have that $a^{(p-1)/2} = u$ for some u which will have order 2, and hence $u = -1$. Putting these two facts together implies we can easily compute the Legendre symbol, via

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod{p}.$$

Using the above formula turns out to be a very inefficient way to compute the Legendre symbol. In practice one uses the *law of quadratic reciprocity*

$$\left(\frac{q}{p}\right) = \left(\frac{p}{q}\right)(-1)^{(p-1)(q-1)/4}. \tag{1}$$

In other words we have

$$\left(\frac{q}{p}\right) = \begin{cases} -\left(\frac{p}{q}\right) & \text{If } p = q = 3 \pmod 4, \\ \left(\frac{p}{q}\right) & \text{Otherwise.} \end{cases}$$

Using this law with the following additional formulae gives rise to a recursive algorithm for the Legendre symbol:

$$\left(\frac{q}{p}\right) = \left(\frac{q \pmod p}{p}\right), \tag{2}$$

$$\left(\frac{q \cdot r}{p}\right) = \left(\frac{q}{p}\right) \cdot \left(\frac{r}{p}\right), \tag{3}$$

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}. \tag{4}$$

Assuming we can factor, we can now compute the Legendre symbol

$$\begin{aligned}
\left(\frac{15}{17}\right) &= \left(\frac{3}{17}\right) \cdot \left(\frac{5}{17}\right) && \text{by equation (3)} \\
&= \left(\frac{17}{3}\right) \cdot \left(\frac{17}{5}\right) && \text{by equation (1)} \\
&= \left(\frac{2}{3}\right) \cdot \left(\frac{2}{5}\right) && \text{by equation (2)} \\
&= (-1) \cdot (-1)^3 && \text{by equation (4)} \\
&= 1.
\end{aligned}$$

In a moment we shall see a more efficient algorithm which does not require us to factor integers.

1.3.6. Computing Square Roots Modulo p: Computing square roots of elements in $\mathbb{F}_p^*$ when the square root exists turns out to be an easy task. Algorithm 1.3 gives one method, called Shanks' Algorithm, of computing the square root of a modulo p, when such a square root exists. When $p = 3 \pmod 4$, instead of the Shank's algorithm, we can use the following formula

$$x \leftarrow a^{(p+1)/4} \pmod p,$$

which has the advantage of being deterministic and more efficient than the general method of Shanks. That this formula works is because

$$x^2 = a^{(p+1)/2} = a^{(p-1)/2} \cdot a = \left(\frac{a}{p}\right) \cdot a = a$$

where the last equality holds since we have assumed that a is a quadratic residue modulo p and so it has Legendre symbol equal to one.

1.3.7. The Jacobi Symbol: The Legendre symbol above is only defined when its denominator is a prime, but there is a generalization to composite denominators called the *Jacobi symbol*. Suppose $n \geq 3$ is odd and

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k},$$

then the Jacobi symbol

$$\left(\frac{a}{n}\right)$$

is defined in terms of the Legendre symbol by

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k}.$$

The Jacobi symbol can be computed using a similar method to the Legendre symbol by making use of the identity, derived from the law of quadratic reciprocity,

$$\left(\frac{a}{n}\right) = \left(\frac{2}{n}\right)^e \cdot \left(\frac{n \pmod{a_1}}{a_1}\right) (-1)^{(a_1-1)\cdot(n-1)/4}.$$

Algorithm 1.3: Shanks' algorithm for extracting a square root of a modulo p

Choose a random n until one is found such that
$$\left(\frac{n}{p}\right) = -1.$$
Let e, q be integers such that q is odd and $p - 1 = 2^e \cdot q$.
$y \leftarrow n^q \pmod{p}$.
$r \leftarrow e$.
$x \leftarrow a^{(q-1)/2} \pmod{p}$.
$b \leftarrow a \cdot x^2 \pmod{p}$.
$x \leftarrow a \cdot x \pmod{p}$.
while $b \neq 1 \pmod{p}$ **do**
  Find the smallest m such that $b^{2^m} = 1 \pmod{p}$.
  $t \leftarrow y^{2^{r-m-1}} \pmod{p}$.
  $y \leftarrow t^2 \pmod{p}$.
  $r \leftarrow m$.
  $x \leftarrow x \cdot t \pmod{p}$.
  $b \leftarrow b \cdot y \pmod{p}$.
return x.

where $a = 2^e \cdot a_1$ and a_1 is odd. We also have the identities, for n odd,
$$\left(\frac{1}{n}\right) = 1, \ \left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}, \ \left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}.$$
This now gives us a fast algorithm, which does not require factoring of integers, to determine the Jacobi symbol, and so the Legendre symbol in the case where the denominator is prime. The only factoring required is to extract the even part of a number. See Algorithm 1.4 which computes the symbol $\left(\frac{a}{b}\right)$. As an example we have
$$\left(\frac{15}{17}\right) = (-1)^{56}\left(\frac{17}{15}\right) = \left(\frac{2}{15}\right) = (-1)^{28} = 1.$$

1.3.8. Squares and Pseudo-squares Modulo a Composite: Recall that the Legendre symbol $\left(\frac{a}{p}\right)$ tells us whether a is a square modulo p, for p a prime. Alas, the Jacobi symbol $\left(\frac{a}{n}\right)$ does not tell us the whole story about whether a is a square modulo n, when n is a composite. If a is a square modulo n then the Jacobi symbol will be equal to plus one, however if the Jacobi symbol is equal to plus one then it is not always true that a is a square.

Let $n \geq 3$ be odd and let the set of squares in $(\mathbb{Z}/n\mathbb{Z})^*$ be denoted by
$$Q_n = \{x^2 \pmod{n} : x \in (\mathbb{Z}/n\mathbb{Z})^*\}.$$
Now let J_n denote the set of elements with Jacobi symbol equal to plus one, i.e.
$$J_n = \left\{x \in (\mathbb{Z}/n\mathbb{Z})^* : \left(\frac{a}{n}\right) = 1\right\}.$$
The set of pseudo-squares is the difference $J_n \setminus Q_n$. There are two important cases for cryptography, either n is prime or n is the product of two primes:

- n is a prime p:
  - $Q_n = J_n$.
  - $\#Q_n = (n-1)/2$.

Algorithm 1.4: Jacobi symbol algorithm

```
if b ≤ 0 or b (mod 2) = 0 then return 0.
j ← 1.
if a < 0 then
    a ← −a.
    if b (mod 4) = 3 then j ← −j.
while a ≠ 0 do
    while a (mod 2) = 0 do
        a ← a/2.
        if b (mod 8) = 3 or b (mod 8) = 5 then j ← −j.
    (a, b) ← (b, a).
    if a (mod 4) = 3 and b (mod 4) = 3 then j ← −j.
    a ← a (mod b).
if b = 1 then return j.
return 0.
```

- n is the product of two primes, $n = p \cdot q$:
  - $Q_n \subset J_n$.
  - $\#Q_n = \#(J_n \setminus Q_n) = (p-1)(q-1)/4$.

The sets Q_n and J_n will be seen to be important in a number of algorithms and protocols, especially in the case where n is a product of two primes.

1.3.9. Square Roots Modulo $n = p \cdot q$: We now look at how to compute a square root modulo a composite number $n = p \cdot q$. Suppose we wish to compute the square root of a modulo n. We assume we know p and q, and that a really is a square modulo n, which can be checked by demonstrating that

$$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = 1.$$

We first compute a square root of a modulo p, call this s_p. Then we compute a square root of a modulo q, call this s_q. Finally to deduce the square root modulo n, we apply the Chinese Remainder Theorem to the equations

$$x = s_p \pmod{p} \text{ and } x = s_q \pmod{q}.$$

Note that if we do not know the prime factors of n then computing square roots modulo n is believed to be a very hard problem; indeed it is as hard as factoring n itself.

As an example suppose we wish to compute the square root of $a = 217$ modulo $n = 221 = 13 \cdot 17$. Now a square root of a modulo 13 and 17 is given by

$$s_{13} = 3 \text{ and } s_{17} = 8.$$

Applying the Chinese Remainder Theorem we find

$$s = 42$$

and we can check that s really is a square root by computing

$$s^2 = 42^2 = 217 \pmod{n}.$$

There are three other square roots, since n has two prime factors. These other square roots are obtained by applying the Chinese Remainder Theorem to the other three equation pairs

$$\begin{aligned} s_{13} &= 10, \quad s_{17} = 8, \\ s_{13} &= 3, \quad s_{17} = 9, \\ s_{13} &= 10, \quad s_{17} = 9, \end{aligned}$$

Hence, all four square roots of 217 modulo 221 are given by 42, 94, 127 and 179.

1.4. Probability

At some points we will need a basic understanding of elementary probability theory. In this section we summarize the theory we require and give a few examples. Most readers should find this a revision of the type of probability encountered in high school. A *random variable* is a variable X which takes certain values with given probabilities. If X takes the value s with probability 0.01 we write this as

$$p(X = s) = 0.01.$$

As an example, let T be the random variable representing tosses of a fair coin, we then have the probabilities

$$\begin{aligned} p(T = \text{Heads}) &= \frac{1}{2}, \\ p(T = \text{Tails}) &= \frac{1}{2}. \end{aligned}$$

As another example let E be the random variable representing letters in English text. An analysis of a large amount of English text allows us to approximate the relevant probabilities by

$$\begin{aligned} p(E = a) &= 0.082, \\ &\vdots \\ p(E = e) &= 0.127, \\ &\vdots \\ p(E = z) &= 0.001. \end{aligned}$$

Basically if X is a discrete random variable on a set S, and $p(X = x)$ is the *probability distribution*, i.e. the probability of a value x being selected from S, then we have the two following properties:

$$\begin{aligned} p(X = x) &\geq 0 \text{ for all } x \in S, \\ \sum_{x \in S} p(X = x) &= 1. \end{aligned}$$

It is common to illustrate examples from probability theory using a standard deck of cards. We shall do likewise and let V denote the random variable that a card is a particular value, let S denote the random variable that a card is a particular suit and let C denote the random variable of the colour of a card. So for example

$$\begin{aligned} p(C = \text{Red}) &= \frac{1}{2}, \\ p(V = \text{Ace of Clubs}) &= \frac{1}{52}, \\ p(S = \text{Clubs}) &= \frac{1}{4}. \end{aligned}$$

Let X and Y be two random variables, where $p(X = x)$ is the probability that X takes the value x and $p(Y = y)$ is the probability that Y takes the value y. The *joint probability* $p(X = x, Y = y)$ is defined as the probability that X takes the value x and Y takes the value y. So if we let $X = C$ and $Y = S$ then we have

$$\begin{aligned}
&p(C = \text{Red}, S = \text{Club}) = 0, && p(C = \text{Red}, S = \text{Diamonds}) = \frac{1}{4},\\
&p(C = \text{Red}, S = \text{Hearts}) = \frac{1}{4}, && p(C = \text{Red}, S = \text{Spades}) = 0,\\
&p(C = \text{Black}, S = \text{Club}) = \frac{1}{4}, && p(C = \text{Black}, S = \text{Diamonds}) = 0,\\
&p(C = \text{Black}, S = \text{Hearts}) = 0, && p(C = \text{Black}, S = \text{Spades}) = \frac{1}{4}.
\end{aligned}$$

Two random variables X and Y are said to be *independent* if, for all values of x and y,

$$p(X = x, Y = y) = p(X = x) \cdot p(Y = y).$$

Hence, the random variables C and S are not independent. As an example of independent random variables consider the two random variables T_1 the value of the first toss of an unbiased coin and T_2 the value of a second toss of the coin. Since, assuming standard physical laws, the toss of the first coin does not affect the outcome of the toss of the second coin, we say that T_1 and T_2 are independent. This is confirmed by the joint probability distribution

$$\begin{aligned}
&p(T_1 = H, T_2 = H) = \frac{1}{4}, && p(T_1 = H, T_2 = T) = \frac{1}{4},\\
&p(T_1 = T, T_2 = H) = \frac{1}{4}, && p(T_1 = T, T_2 = T) = \frac{1}{4}.
\end{aligned}$$

1.4.1. Bayes' Theorem: The *conditional probability* $p(X = x \mid Y = y)$ of two random variables X and Y is defined as the probability that X takes the value x given that Y takes the value y. Returning to our random variables based on a pack of cards we have

$$p(S = \text{Spades} \mid C = \text{Red}) = 0$$

and

$$p(V = \text{Ace of Spades} \mid C = \text{Black}) = \frac{1}{26}.$$

The first follows since if we know that a card is red, then the probability that it is a spade is zero, since a red card cannot be a spade. The second follows since if we know a card is black then we have restricted the set of cards to half the pack, one of which is the ace of spades.

The following is one of the most crucial statements in probability theory, which you should recall from high school,

Theorem 1.7 (Bayes' Theorem)**.** *If* $p(Y = y) > 0$ *then*

$$\begin{aligned}
p(X = x|Y = y) &= \frac{p(X = x) \cdot p(Y = y \mid X = x)}{p(Y = y)}\\
&= \frac{p(X = x, Y = y)}{p(Y = y)}.
\end{aligned}$$

We can apply Bayes' Theorem to our examples above as follows

$$\begin{aligned}
p(S = \text{Spades} \mid C = \text{Red}) &= \frac{p(S = \text{Spades}, C = \text{Red})}{p(C = \text{Red})}\\
&= 0 \cdot \left(\frac{1}{4}\right)^{-1} = 0.
\end{aligned}$$

$$\begin{aligned} p(V = \text{Ace of Spades} \mid C = \text{Black}) &= \frac{p(V = \text{Ace of Spades}, C = \text{Black})}{p(C = \text{Black})} \\ &= \frac{1}{52} \cdot \left(\frac{1}{2}\right)^{-1} \\ &= \frac{2}{52} = \frac{1}{26}. \end{aligned}$$

If X and Y are independent then we have

$$p(X = x|Y = y) = p(X = x),$$

i.e. the value that X takes does not depend on the value that Y takes. An identity which we will use a lot is the following, for events A and B

$$\begin{aligned} p(A) &= p(A, B) + p(A, \neg B) \\ &= p(A|B) \cdot p(B) + p(A|\neg B) \cdot p(\neg B). \end{aligned}$$

where $\neg B$ is the event that B does not happen.

1.4.2. Birthday Paradox: Another useful result from elementary probability theory that we will require is the *birthday paradox*. Suppose a bag has m balls in it, all of different colours. We draw one ball at a time from the bag and write down its colour, we then replace the ball in the bag and draw again. If we define

$$m^{(n)} = m \cdot (m-1) \cdot (m-2) \cdots (m-n+1)$$

then the probability, after n balls have been taken out of the bag, that we have obtained at least one matching colour (or coincidence) is

$$1 - \frac{m^{(n)}}{m^n}.$$

As m becomes larger the expected number of balls we have to draw before we obtain the first coincidence is

$$\sqrt{\frac{\pi \cdot m}{2}}.$$

To see why this is called the birthday paradox consider the probability of two people in a room sharing the same birthday. Most people initially think that this probability should be quite low, since they are thinking of the probability that someone in the room shares the same birthday as them. One can now easily compute that the probability of at least two people in a room of 23 people having the same birthday is

$$1 - \frac{365^{(23)}}{365^{23}} \approx 0.507.$$

In fact this probability increases quite quickly since in a room of 30 people we obtain a probability of approximately 0.706, and in a room of 100 people we obtain a probability of over 0.999 999 6.

In many situations in cryptography we use the birthday paradox in the following way. We are given a random process which outputs elements from a set of size m, just like the balls above. We run the process for n steps, again just like above. But instead of wanting to know how many times we need to execute the process to find a collision we instead want to know an upper bound on the probability of finding a collision after n steps (think of n being much smaller than m). This is easy

to estimate due to the following inequalities:

$$\begin{aligned}\Pr\left[\text{At least one repetition in pulling } n \text{ elements from } m\right] &\leq \sum_{1\leq i<j<n} \Pr[\text{Item } i \text{ collides with item } j] \\ &= \binom{n}{2} \cdot \frac{1}{m} \\ &= \frac{n\cdot(n-1)}{m} \leq \frac{n^2}{2\cdot m}.\end{aligned}$$

1.5. Big Numbers

At various points we need to discuss how big a number can be before it is impossible for someone to perform that many operations. Such big numbers are used in cryptography to measure the work effort of the adversary. Suppose we had a (mythical) computer which could do one trillion "basic" operations per second. Note that a modern 3 GHz computer with eight "cores" can only do 24 billion operations per second, so our mythical computer is around 42 times faster than a current desktop computer.

Suppose we had an algorithm which took 2^t "basic" operations. We want to know how long our mythical computer would take to perform these 2^t operations. Now one trillion is about 2^{40}. Thus to perform 2^{64} operations would require $2^{64-40} = 2^{24}$ seconds, or 194 days. Given that finding 194 computers is not very hard, a calculation which takes 2^{64} basic operations could be performed by someone with just under 200 computers in under a day. An algorithm which took 2^{80} "basic" operations would take 2^{40} seconds for our mythical computer, or nearly 34 900 years. Thus a large government-funded laboratory which could afford perhaps 15 000 mythical computers could perform the algorithm requiring 2^{80} operations in about two years. This might be expensive, but if national security depended on it, then a computation of 2^{80} operations would be plausible.

However, when we go to an algorithm which requires 2^{128} operations then our mythical computer would require 2^{88} seconds or 9 quintillion years (i.e. $9\cdot10^{18}$ years). Note, the universe is only believed to be 13.8 billion years old. Thus a computation which required 9 quintillion years is essentially impossible, ever!!!!

To get an idea of how big these numbers are consider that 2^{80} is a number with 24 decimal digits, whereas 2^{128} is a number with 38 decimal digits. These are both significantly more than the number of cells in the human body (which is around 10^{14}), or the number of stars in the observable universe (which is around 10^{22}).

Chapter Summary

- A group is a set with an operation which has an identity, is associative and in which every element has an inverse.
- Addition and multiplication in modular arithmetic both provide examples of groups.
- For modular multiplication we need to be careful which set of numbers we take when defining such a group, as not all integers modulo m are invertible with respect to multiplication.

- A ring is a set with two operations which behaves like the set of integers under addition and multiplication. Modular arithmetic is an example of a ring.
- A field is a ring in which all non-zero elements have a multiplicative inverse. The integers modulo a prime is an example of a field.
- Multiplicative inverses for modular arithmetic can be found using the extended Euclidean algorithm.
- Sets of simultaneous linear modular equations can be solved using the Chinese Remainder Theorem.
- Square elements modulo a prime can be detected using the Legendre symbol; square roots can be efficiently computed using Shanks' Algorithm.
- Square elements and square roots modulo a composite can be determined efficiently as long as one knows the factorization of the modulus.
- Bayes' Theorem allows us to compute conditional probabilities.
- The birthday paradox allows us to estimate how quickly collisions occur when one repeatedly samples from a finite space.
- We also discussed how big various numbers are, as a means to work out what is a feasible computation.

Further Reading

Bach and Shallit is the best introductory book I know of which deals with Euclid's algorithm and finite fields. It contains a lot of historical information, plus excellent pointers to the relevant research literature. Whilst aimed in some respects at Computer Scientists, Bach and Shallit's book may be a little too mathematical for some. For a more traditional introduction to the basic discrete mathematics we shall need, see the books by Biggs or Rosen.

E. Bach and J. Shallit. *Algorithmic Number Theory. Volume 1: Efficient Algorithms*. MIT Press, 1996.

N.L. Biggs. *Discrete Mathematics*. Oxford University Press, 1989.

K.H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 1999.

CHAPTER 2

Primality Testing and Factoring

Chapter Goals

- To explain the basics of primality testing.
- To describe the most used primality-testing algorithm, namely Miller–Rabin.
- To examine the relationship between various mathematical problems based on factoring.
- To explain various factoring algorithms.
- To sketch how the most successful factoring algorithm works, namely the Number Field Sieve.

2.1. Prime Numbers

The generation of prime numbers is needed for almost all public key algorithms, for example

- In the RSA encryption or the Rabin encryption system we need to find primes p and q to compute the public key $N = p \cdot q$.
- In ElGamal encryption we need to find primes p and q with q dividing $p - 1$.
- In the elliptic curve variant of ElGamal we require an elliptic curve over a finite field, such that the order of the elliptic curve is divisible by a large prime q.

Luckily we shall see that testing a number for primality can be done very fast using very simple code, but with an algorithm that has a probability of error. By repeating this algorithm we can reduce the error probability to any value that we require.

Some of the more advanced primality-testing techniques will produce a certificate which can be checked by a third party to prove that the number is indeed prime. Clearly one requirement of such a certificate is that it should be quicker to verify than it is to generate. Such a primality-testing routine will be called a primality-proving algorithm, and the certificate will be called a proof of primality. However, the main primality-testing algorithm used in cryptographic systems only produces certificates of compositeness and not certificates of primality.

For many years this was the best that we could do; i.e. either we could use a test which had a small chance of error, or we spent a lot of time producing a proof of primality which could be checked quickly. However, in 2002 Agrawal, Kayal and Saxena presented a deterministic polynomial-time primality test thus showing that the problem of determining whether a number was prime was in the complexity class $\mathcal{P}$. However, the so-called AKS Algorithm is not used in practice as the algorithms which have a small error are more efficient and the error can be made vanishingly small at little extra cost.

2.1.1. The Prime Number Theorem: Before discussing these algorithms, we need to look at some basic heuristics concerning prime numbers. A famous result in mathematics, conjectured by Gauss after extensive calculation in the early 1800s, is the Prime Number Theorem:

Theorem 2.1 (Prime Number Theorem)**.** *The function $\pi(X)$ counts the number of primes less than X, where we have the approximation*

$$\pi(X) \approx \frac{X}{\log X}.$$

This means primes are quite common. For example, the number of primes less than 2^{1024} is about 2^{1014}. The Prime Number Theorem also allows us to estimate the probability of a random number being prime: if p is a number chosen at random then the probability it is prime is about

$$\frac{1}{\log p}.$$

So a random number p of 1024 bits in length will be a prime with probability

$$\approx \frac{1}{\log p} \approx \frac{1}{709}.$$

So on average we need to select 354 odd numbers of size 2^{1024} before we find one which is prime. Hence, it is practical to generate large primes, as long as we can test primality efficiently.

2.1.2. Trial Division: The naive test for testing a number p to be prime is one of trial division. We essentially take all numbers between 2 and $\sqrt{p}$ and see whether one of them divides p, if not then p is prime. If such a number does divide p then we obtain the added bonus of finding a factor of the composite number p. Hence, trial division has the advantage (compared with more advanced primality-testing/proving algorithms) that it either determines that p is a prime, or determines a non-trivial factor of p.

However, primality testing by using trial division is a terrible strategy. In the worst case, when p is a prime, the algorithm requires $\sqrt{p}$ steps to run, which is an exponential function in terms of the size of the input to the problem. Another drawback is that it does not produce a certificate for the primality of p, in the case when the input p is prime. When p is not prime it produces a certificate which can easily be checked to prove that p is composite, namely a non-trivial factor of p. But when p is prime the only way we can verify this fact again (say to convince a third party) is to repeat the algorithm once more.

Despite its drawbacks, however, trial division is the method of choice for numbers which are very small. In addition, partial trial division up to a bound Y is able to eliminate all but a proportion

$$\prod_{p<Y} \left(1 - \frac{1}{p}\right)$$

of all composites. This method of eliminating composites is very old and is called the Sieve of Eratosthenes. Naively this is what we would always do, since we would never check an even number greater than two for primality, since it is obviously composite. Hence, many primality-testing algorithms first do trial division with all primes up to say 100, so as to eliminate all but the proportion

$$\prod_{p<100} \left(1 - \frac{1}{p}\right) \approx 0.12$$

of composites.

2.1.3. Fermat's Test: Most advanced probabilistic algorithms for testing primality make use of the converse to Fermat's Little Theorem. Recall Lagrange's Theorem from Chapter 1; this said that if G is a multiplicative group of size $\#G$ then

$$a^{\#G} = 1$$

for all values $a \in G$. So if G is the group of integers modulo n under multiplication then

$$a^{\phi(n)} = 1 \pmod{n}$$

for all $a \in (\mathbb{Z}/n\mathbb{Z})^*$. Fermat's Little Theorem is the case where $n = p$ is prime, in which case the above equality becomes

$$a^{p-1} = 1 \pmod{p}.$$

So if n is prime we have that

$$a^{n-1} = 1 \pmod{n}$$

always holds, whilst if n is not prime then we have that

$$a^{n-1} = 1 \pmod{n}$$

is "unlikely" to hold.

Since computing $a^{n-1} \pmod{n}$ is a very fast operation (see Chapter 6) this gives us a very fast test for compositeness called the Fermat Test to the base a. Running the Fermat Test can only convince us of the compositeness of n. It can never prove to us that a number is prime, only that it is not prime.

To see why it does not prove primality consider the case $n = 11 \cdot 31 = 341$ and the base $a = 2$: we have

$$a^{n-1} = 2^{340} = 1 \pmod{341}.$$

but n is clearly not prime. In such a case we say that n is a (Fermat) pseudo-prime to the base 2. There are infinitely many pseudo-primes to any given base. It can be shown that if n is composite then, with probability greater than $1/2$, we obtain

$$a^{n-1} \neq 1 \pmod{n}.$$

This gives us Algorithm 2.1 to test n for primality. If Algorithm 2.1 outputs (Composite, a) then

Algorithm 2.1: Fermat's test for primality

for $i = 0$ **to** $k - 1$ **do**
  Pick $a \in [2, ..., n-1]$.
  $b \leftarrow a^{n-1} \mod n$.
  if $b \neq 1$ **then return** (Composite, a).
return "Probably Prime".

we know

- n is definitely a composite number,
- a is a witness for this compositeness, in that we can verify that n is composite by using the value of a.

If the above algorithm outputs "Probably Prime" then

- n is a composite with probability at most $1/2^k$,
- n is either a prime or a so-called probable prime.

For example if we take

$$n = 43\,040\,357,$$

then n is a composite, with one witness given by $a = 2$ since

$$2^{n-1} \pmod{n} = 9\,888\,212.$$

As another example take

$$n = 2^{192} - 2^{64} - 1,$$

then the algorithm outputs "Probably Prime" since we cannot find a witness for compositeness. Actually this n is a prime, so it is not surprising we did not find a witness for compositeness!

However, there are composite numbers for which the Fermat Test will always output

"Probably Prime"

for every a coprime to n. These numbers are called Carmichael numbers, and to make things worse there are infinitely many of them. The first three are $561, 1105$ and 1729. Carmichael numbers have the following properties

- They are always odd.
- They have at least three prime factors.
- They are square free.
- If p divides a Carmichael number N, then $p-1$ divides $N-1$.

To give you some idea of their density, if we look at all numbers less than 10^{16} then there are about $2.7 \cdot 10^{14}$ primes in this region, but only $246\,683 \approx 2.4 \cdot 10^5$ Carmichael numbers. Hence, Carmichael numbers are rare, but not rare enough to be ignored completely.

2.1.4. Miller–Rabin Test: Due to the existence of Carmichael numbers the Fermat Test is usually avoided. However, there is a modification of the Fermat Test, called the Miller–Rabin Test, which avoids the problem of composites for which no witness exists. This does not mean it is easy to find a witness for each composite, it only means that a witness must exist. In addition the Miller–Rabin Test has probability of $1/4$ of accepting a composite as prime for each random base a, so again repeated application of the algorithm leads us to reduce the error probability down to any value we care to mention.

The Miller–Rabin Test is given by the pseudo-code in Algorithm 2.2. We do not show that the Miller–Rabin Test works. If you are interested in the reason see any book on algorithmic number theory for the details, for example that by Cohen or Bach and Shallit mentioned in the Further Reading section of this chapter. Just as with the Fermat Test, we repeat the method k times with k different bases, to obtain an error probability of $1/4^k$ if the algorithm always returns "Probably Prime". Hence, we expect that the Miller–Rabin Test will output "Probably Prime" for values of $k \geq 20$ only when n is actually a prime.

Algorithm 2.2: Miller–Rabin algorithm

```
Write n − 1 = 2^s · m, with m odd.
for j = 0 to k − 1 do
    Pick a ∈ [2, ..., n − 2].
    b ← a^m mod n.
    if b ≠ 1 and b ≠ (n − 1) then
        i ← 1.
        while i < s and b ≠ (n − 1) do
            b ← b^2 mod n.
            if b = 1 then return (Composite, a).
            i ← i + 1.
        if b ≠ (n − 1) then return (Composite, a).
return "Probable Prime".
```

If n is a composite then the value of a output by Algorithm 2.2 is called a Miller–Rabin witness for the compositeness of n, and under the Generalized Riemann Hypothesis (GRH), a conjecture

believed to be true by most mathematicians, there is always a Miller–Rabin witness a for the compositeness of n with

$$a \leq O((\log n)^2).$$

2.1.5. Primality Proofs: Up to now we have only output witnesses for compositeness, and we can interpret such a witness as a proof of compositeness. In addition we have only obtained probable primes, rather than numbers which are one hundred percent guaranteed to be prime. In practice this seems to be all right, since the probability of a composite number passing the Miller–Rabin Test for twenty bases is around 2^{-40} which should never really occur in practice. But theoretically (and maybe in practice if we are totally paranoid) this could be a problem. In other words we may want real primes and not just probable ones.

There are algorithms whose output is a witness for the primality of the number. Such a witness is called a proof of primality. In practice such programs are only used when we are morally certain that the number we are testing for primality is actually prime. In other words the number has already passed the Miller–Rabin Test for a number of bases and all we now require is a proof of the primality.

The most successful of these primality-proving algorithms is one based on elliptic curves called ECPP (for Elliptic Curve Primality Prover). This itself is based on an older primality-proving algorithm based on finite fields due to Pocklington and Lehmer; the elliptic curve variant is due to Goldwasser and Kilian. The ECPP algorithm is a randomized algorithm which is not mathematically guaranteed to always produce an output, i.e. a witness, even when the input is a prime number. If the input is composite then the algorithm is not guaranteed to terminate at all. Although ECPP runs in expected polynomial time, i.e. it is quite efficient, the proofs of primality it produces can be deterministically verified even faster.

There is an algorithm due to Adleman and Huang which, unlike the ECPP method, is guaranteed to terminate with a proof of primality on input of a prime number. It is based on a generalization of elliptic curves called hyperelliptic curves and has never (to my knowledge) been implemented. The fact that it has never been implemented is not only due to the far more complicated mathematics involved, but is also due to the fact that while the hyperelliptic variant is mathematically guaranteed to produce a proof, the ECPP method will always do so in practice for less work effort.

2.1.6. AKS Algorithm: The Miller–Rabin Test is a randomized primality-testing algorithm which runs in polynomial time. It can be made into a deterministic polynomial-time algorithm, but only on the assumption that the Generalized Riemann Hypothesis is true. The ECPP algorithm and its variants are randomized algorithms and are expected to have polynomial-time run-bounds, but we cannot prove they do so on all inputs. Thus for many years it was an open question whether we could create a primality-testing algorithm which ran in *deterministic* polynomial time, and provably so on all inputs without needing to assume any conjectures. In other words, the question was whether the problem PRIMES is in complexity class $\mathcal{P}$?

In 2002 this was answered in the affirmative by Agrawal, Kayal, and Saxena. The test they developed, now called the AKS Primality Test, makes use of the following generalization of Fermat's test. In the theorem we are asking whether two polynomials of degree n are the same. Taking this basic theorem, which is relatively easy to prove, and turning it into a polynomial-time test was a major breakthrough. The algorithm itself is given in Algorithm 2.3. In the algorithm we use the notation $F(X) \pmod{G(X), n}$ to denote taking the reduction of $F(X)$ modulo *both* $G(X)$ and n.

Theorem 2.2. *An integer $n \geq 2$ is prime if and only if the relation*

$$(X - a)^n = (X^n - a) \pmod{n}$$

holds for some integer a coprime to n; or indeed all integers a coprime to n.

Algorithm 2.3: AKS primality-testing algorithm

if $n = a^b$ for some integers a and b **then return** "Composite".
Find the smallest r such that the order of n modulo r is greater than $(\log n)^2$.
if $\exists a \leq r$ such that $1 < \gcd(a, n) < n$ **then return** "Composite".
if $n \leq r$ **then return** "Prime".
for $a = 1$ **to** $\lfloor \sqrt{\phi(r)} \cdot \log(n) \rfloor$ **do**
  if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$ **then return** "Composite"
return Prime

2.2. The Factoring and Factoring-Related Problems

The most important one-way function used in public key cryptography is that of factoring integers. By factoring an integer we mean finding its prime factors, for example

$$10 = 2 \cdot 5,$$
$$60 = 2^2 \cdot 3 \cdot 5,$$
$$2^{113} - 1 = 3391 \cdot 23\,279 \cdot 65\,993 \cdot 1\,868\,569 \cdot 1\,066\,818\,132\,868\,207.$$

There are a number of other hard problems related to factoring which can be used to produce public key cryptosystems. Suppose you are given an integer N, which is known to be the product of two large primes, but not its factors p and q. There are four main problems which we can try to solve:

- **FACTOR:** Find p and q.
- **RSA:** Given e such that
$$\gcd(e, (p-1)(q-1)) = 1$$
and c, find m such that
$$m^e = c \pmod{N}.$$
- **SQRROOT:** Given a such that
$$a = x^2 \pmod{N},$$
find x.
- **QUADRES:** Given $a \in J_N$, determine whether a is a square modulo N.

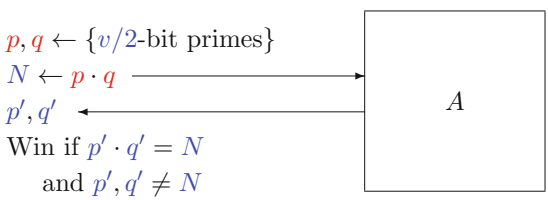


FIGURE 2.1. Security game to define the FACTOR problem

In Chapter 11, we use so-called security games to define security for cryptographic components. These are abstract games played between an adversary and a challenger. The idea is that the adversary needs to achieve some objective given only the data provided by the challenger. Such games tend to be best described using pictures, where the challenger (or environment) is listed on the outside and the adversary is presented as a box. The reason for using such diagrams will

become clearer later when we consider security proofs, but for now they are simply going to be used to present security definitions.

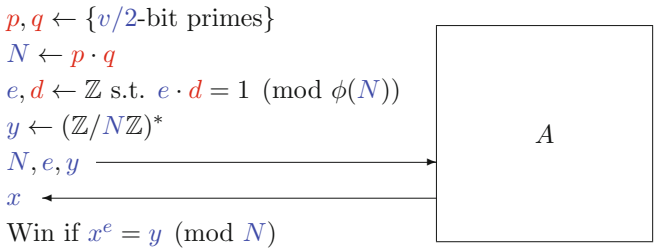

$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
$e, d \leftarrow \mathbb{Z}$ s.t. $e \cdot d = 1 \pmod{\phi(N)}$
$y \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$
N, e, y
x
Win if $x^e = y \pmod{N}$
A

FIGURE 2.2. Security game to define the RSA problem

So for example, we could imagine a game which defines the problem of an adversary A trying to factor a challenge number N as in Figure 2.1. The challenger comes up with two secret prime numbers, multiplies them together and sends the product to the adversary. The adversary's goal is to find the original prime numbers. Similarly we can define games for the RSA and SQRROOT problems, which we give in Figures 2.2 and 2.3.

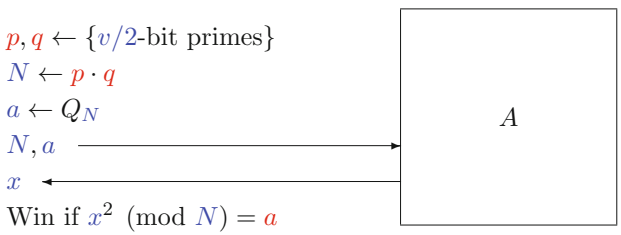


FIGURE 2.3. Security game to define the SQRROOT problem

In all these games we define the advantage of a specific adversary A to be a function of the time t which the adversary spends trying to solve the input problem. For the Factoring, RSA and SQRROOT games it is defined as the probability (defined over the random choices made by A) that the adversary wins the game given that it runs in time bounded by t (we are not precise on what units t is measured in). We write

$$\mathrm{Adv}_v^X(A, t) = \Pr[A \text{ wins the game } X \text{ for } v = \log_2 N \text{ in time less than } t].$$

If the adversary is always successful then the advantage will be one, if the adversary is never successful then the advantage will be zero.

In the next section we will see that there is a trivial algorithm which always factors a number in time $\sqrt{N}$. So we know that *there is* an adversary A such that

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A, 2^{v/2}) = 1.$$

However if t is any *polynomial* function p_1 of $v = \log_2 N$ then we expect that there is no efficient adversary A, and hence for such t we will have

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A, p_1(v)) < \frac{1}{p_2(v)},$$

for *any* polynomial $p_2(x)$ and *for all* adversaries A. A function which grows less quickly than $1/p_2(x)$ for any polynomial function of $p_2(x)$ is said to be *negligible*, so we say the advantage of

solving the factoring problem is negligible. Note that, even if the game was played again and again (but a polynomial in v number of times), the adversary would still obtain a negligible probability of winning since a negligible function multiplied by a polynomial function is still negligible.

In the rest of this book we will drop the time parameter from the advantage statement and implicitly assume that all adversaries run in polynomial time; thus we simply write $\mathrm{Adv}_Y^X(A)$, $\mathrm{Adv}_v^{\mathsf{FACTOR}}(A)$, $\mathrm{Adv}_v^{\mathsf{RSA}}(A)$ and $\mathrm{Adv}_v^{\mathsf{SQRROOT}}(A)$. We call the subscript the problem class; in the above this is the size v of the composite integers, in Chapter 3 it will be the underlying abelian group. The superscript defines the precise game which the adversary A is playing.

A game X for a problem class Y is said to be *hard* if the advantage is a negligible function *for all* polynomial-time adversaries A. The problem with this definition is that the notion of negligible is asymptotic, and when we consider cryptosystems we usually talk about concrete parameters; for example the fixed size of integers which are to be factored.

Thus, instead, we will deem a class of problems Y to be hard if for all polynomial-time adversaries A, the advantage $\mathrm{Adv}_Y^X(A)$ is a very small value ϵ; think of ϵ as being $1/2^{128}$ or some such number. This means that even if the run time of the adversary was one time unit, and we repeatedly ran the adversary a large number of times, the advantage that the adversary would gain would still be very very small. In this chapter we leave aside the issue of how small "small" is, but in later chapters we examine this in more detail.

The QUADRES problem is a little different as we need to define the probability distribution from which the challenge numbers a come. The standard definition is for the challenger to pick a to be a quadratic residue with probability 1/2. In this way the adversary has a fifty-fifty chance of simply guessing whether a is a quadratic residue or not. We present the game in Figure 2.4.

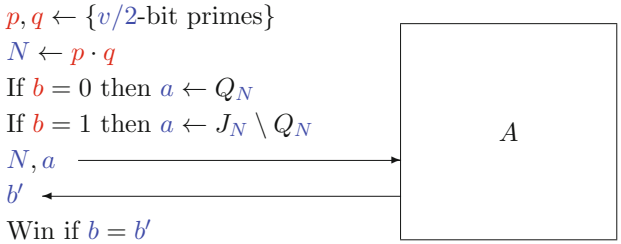

$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
If $b = 0$ then $a \leftarrow Q_N$
If $b = 1$ then $a \leftarrow J_N \setminus Q_N$
N, a
b'
Win if $b = b'$
A

FIGURE 2.4. Security game to define the QUADRES problem

When defining the advantage for the QUADRES problem we need to be a bit careful, as the adversary can always win with probability one half by simply just guessing the bit b at random. Instead of using the above definition of advantage (i.e. the probability that the adversary wins the game), we use the definition

$$\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) = 2 \cdot \left| \Pr[A \text{ wins the QUADRES game for } v = \log_2 N] - \frac{1}{2} \right|.$$

Notice that, with this definition, if the adversary just guesses the bit with probability 1/2 then its advantage is zero as we would expect, since $2 \cdot |1/2 - 1/2| = 0$. If however the adversary is always right, or indeed always wrong, then the advantage is one, since $2 \cdot |1 - 1/2| = 2 \cdot |0 - 1/2| = 1$. Thus the advantage is normalized to lie between zero and one, like in the earlier games, with one being always successful and zero being no better than random.

We call this type of game a *decision game* as the adversary needs to decide which situation it is being placed in. We can formulate the advantage statement for decision games in another way, as the following lemma explains.

Lemma 2.3. *Let A be an adversary in the QUADRES game. Then, if b' is the bit chosen by A and b is the bit chosen by the challenger in the game, we have*

$$\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) = \left|\Pr[b' = 1 | b = 1] - \Pr[b' = 1 | b = 0]\right|.$$

Proof. The proof is a straightforward application of definitions of probabilities:

$$\begin{aligned}
\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) &= 2 \cdot \left|\Pr[A \text{ wins}] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1 \text{ and } b = 1] + \Pr[b' = 0 \text{ and } b = 0] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1 | b = 1] \cdot \Pr[b = 1] + \Pr[b' = 0 | b = 0] \cdot \Pr[b = 0] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1 | b = 1] \cdot \frac{1}{2} + \Pr[b' = 0 | b = 0] \cdot \frac{1}{2} - \frac{1}{2}\right| \\
&= \left|\Pr[b' = 1 | b = 1] + \Pr[b' = 0 | b = 0] - 1\right| \\
&= \left|\Pr[b' = 1 | b = 1] + \big(1 - \Pr[b' = 1 | b = 0]\big) - 1\right| \\
&= \left|\Pr[b' = 1 | b = 1] - \Pr[b' = 1 | b = 0]\right|.
\end{aligned}$$

□

To see how this Lemma works consider the case when A is a perfect adversary, i.e. it wins the QUADRES game all the time. In this case we have $\Pr[A \text{ wins}] = 1$, and the advantage is equal to $2 \cdot |1 - 1/2| = 1$ by definition. However, in this case we also have $\Pr[b' = 1 | b = 1] = 1$ and $\Pr[b' = 1 | b = 0] = 0$. Hence, the formula from the Lemma holds. Now examine what happens when A just returns a random result. We obtain $\Pr[A \text{ wins}] = 1/2$, and the advantage is equal to $2 \cdot |1/2 - 1/2| = 0$. The Lemma gives the same result as $\Pr[b' = 1 | b = 1] = \Pr[b' = 1 | b = 0] = 1/2$.

When giving these problems it is important to know how they are related. We relate them by giving complexity-theoretic reductions from one problem to another. This allows us to say that "Problem B is no harder than Problem A". Assuming an oracle (or efficient subroutine) to solve Problem A, we create an efficient algorithm for Problem B. The algorithms which perform these reductions should be efficient, in that they run in polynomial time, where we treat each oracle query as a single time unit.

We can also show *equivalence* between two problems A and B, by showing an efficient reduction from A to B and an efficient reduction from B to A. If the two reductions are both polynomial-time reductions then we say that the two problems are *polynomial-time equivalent*. The most important result of this form for our factoring related problems is the following.

Theorem 2.4. *The FACTOR and SQRROOT problems are polynomial-time equivalent.*

The next two lemmas present reductions in both directions. By examing the proofs it is easy to see that both of the reductions can be performed in expected polynomial time. Hence, the problems FACTOR and SQRROOT are polynomial-time equivalent. First, in the next lemma, we show how to reduce SQRROOT to FACTOR; if there is no algorithm which can solve SQRROOT then there is no algorithm to solve FACTOR.

Lemma 2.5. *If A is an algorithm which can factor integers of size v, then there is an efficient algorithm B which can solve SQRROOT for integers of size v. In particular*

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A) = \mathrm{Adv}_v^{\mathsf{SQRROOT}}(B).$$

PROOF. Assume we are given a factoring algorithm A; we wish to show how to use this to extract square roots modulo a composite number N. Namely, given

$$a = x^2 \pmod{N}$$

we wish to compute x. First we factor N into its prime factors $p_1, p_2, \ldots, p_k$, using the factoring oracle A. Then we compute

$$s_i \leftarrow \sqrt{a} \pmod{p_i} \text{ for } 1 \leq i \leq k.$$

This can be done in expected polynomial time using Shanks' Algorithm (Algorithm 1.3 from Chapter 1). Then we compute the value of x using the Chinese Remainder Theorem on the data

$$(s_1, p_1), \ldots, (s_k, p_k).$$

We have to be a little careful if powers of p_i greater than one divide N. However, this is easy to deal with and will not concern us here, since we are mainly interested in integers N which are the product of two primes. Hence, finding square roots modulo N is no harder than factoring.

The entire proof can be represented diagramatically in terms of our game diagrams as in Figure 2.5; where we have specialized the game to one of integers N which are the product of two prime factors.

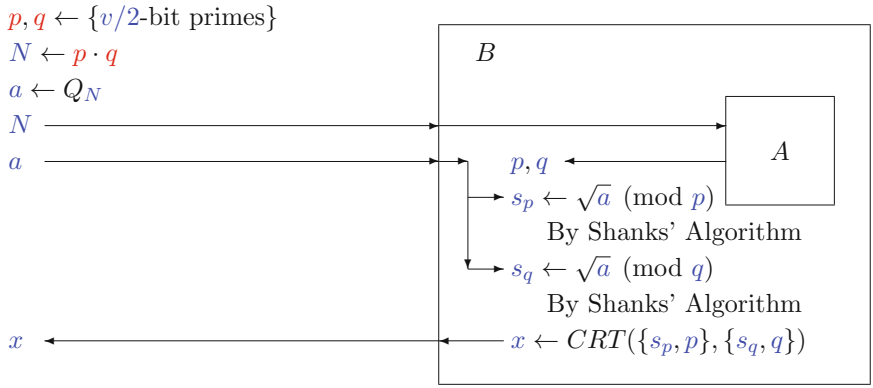


FIGURE 2.5. Constructing an algorithm B to solve SQRROOT from an algorithm A to solve FACTOR

□

We now show how to reduce FACTOR to SQRROOT; if there is no algorithm which can solve FACTOR then there is no algorithm to solve SQRROOT.

Lemma 2.6. *Let A be an algorithm which can solve SQRROOT for integers of size v; then there is an efficient algorithm B which can factor integers of size v. In particular for N a product of two primes we have*

$$\mathrm{Adv}_v^{\mathsf{SQRROOT}}(A) = 2 \cdot \mathrm{Adv}_v^{\mathsf{FACTOR}}(B).$$

The proof of this result contains an important tool used in the factoring algorithms of the next section, namely the construction of a difference of two squares.

PROOF. Assume we are given an algorithm A for extracting square roots modulo a composite number N. We shall assume for simplicity that N is a product of two primes, which is the most difficult case. The general case is only slightly more tricky mathematically, but it is computationally

easier since factoring numbers with three or more prime factors is usually easier than factoring numbers with two prime factors.

We wish to use our algorithm A for the problem SQRROOT to factor the integer N into its prime factors, i.e. given $N = p \cdot q$ we wish to compute p. First we pick a random $x \in (\mathbb{Z}/N\mathbb{Z})^*$ and compute

$$a \leftarrow x^2 \pmod{N}.$$

Now we compute

$$y \leftarrow \sqrt{a} \pmod{N}$$

using the SQRROOT algorithm. There are four such square roots, since N is a product of two primes. With fifty percent probability we obtain

$$y \neq \pm x \pmod{N}.$$

If we do not obtain this inequality then we abort.

We now assume that the inequality holds, but we note that we have the equality $x^2 = y^2 \pmod{N}$. It is then easy to see that N divides

$$x^2 - y^2 = (x - y)(x + y).$$

But N does not divide either $x - y$ or $x + y$, since $y \neq \pm x \pmod{N}$. So the factors of N must be distributed over $x - y$ and $x + y$. This means we can obtain a non-trivial factor of N by computing $\gcd(x - y, N)$

It is because of the above fifty percent probability that we get a factor of two in our advantage statement, since B is only successful if A is successful *and* we obtain $y \neq \pm x \pmod{N}$. Thus $\Pr[B \text{ wins}] = \Pr[A \text{ wins}]/2$. Diagrammatically we represent this reduction in Figure 2.6.

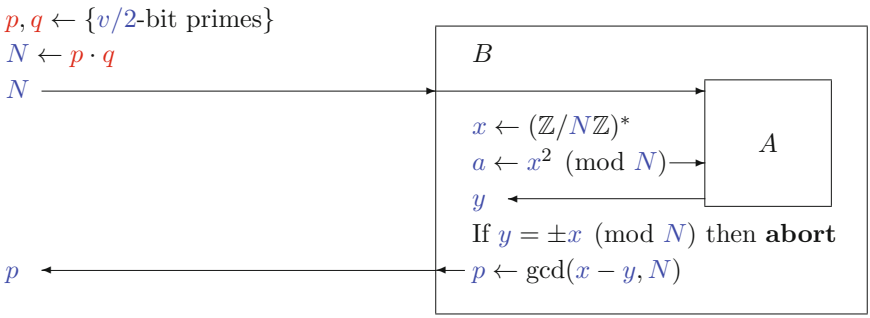


FIGURE 2.6. Constructing an algorithm B to solve FACTOR from an algorithm A to solve SQRROOT

□

Before leaving the problem SQRROOT, note that QUADRES is easier than SQRROOT, since an algorithm to compute square roots modulo N can trivially be used to determine quadratic residuosity.

Finally we end this section by showing that the RSA problem can be reduced to FACTOR. Recall the RSA problem is given $c = m^e \pmod{N}$, find m. There is some evidence, although slight, that the RSA problem may actually be easier than FACTOR for some problem instances. It is a major open question as to how much easier it is.

Lemma 2.7. *The RSA problem is no harder than the FACTOR problem. In particular, if A is an algorithm which can solve FACTOR for integers of size v, then there is an efficient algorithm B which can solve the RSA problem for integers of size v. In particular for N a product of two primes we have*

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A) = \mathrm{Adv}_v^{\mathsf{RSA}}(B).$$

PROOF. Using the factoring algorithm A we first find the factorization of N. We can now compute $\Phi = \phi(N)$ and then compute

$$d \leftarrow 1/e \pmod{\Phi}.$$

Once d has been computed it is easy to recover m via

$$c^d = m^{e\cdot d} = m^{1 \pmod{\Phi}} = m \pmod{N},$$

with the last equality following by Lagrange's Theorem, Theorem 1.4. Hence, the RSA problem is no harder than FACTOR. We leave it to the reader to present a diagram of this reduction similar to the ones above. □

2.3. Basic Factoring Algorithms

Finding factors is an expensive computational operation. To measure the complexity of algorithms to factor an integer N we often use the function

$$L_N(\alpha, \beta) = \exp\left((\beta + o(1))(\log N)^{\alpha}(\log\log N)^{1-\alpha}\right).$$

Note that

- $L_N(0, \beta) = (\log N)^{\beta+o(1)}$, i.e. essentially polynomial time,
- $L_N(1, \beta) = N^{\beta+o(1)}$, i.e. essentially exponential time.

So in some sense, the function $L_N(\alpha, \beta)$ interpolates between polynomial and exponential time. An algorithm with complexity $O(L_N(\alpha, \beta))$ for $0 < \alpha < 1$ is said to have sub-exponential behaviour. Note that multiplication, which is the inverse algorithm to factoring, is a very simple operation requiring time less than $O(L_N(0, 2))$.

There are a number of methods to factor numbers of the form

$$N = p \cdot q.$$

For now we just summarize the most well-known techniques.

- **Trial Division:** Try every prime number up to $\sqrt{N}$ and see whether it is a factor of N. This has complexity $L_N(1, 1)$, and is therefore an exponential algorithm.
- **Elliptic Curve Method:** This is a very good method if $p < 2^{50}$; its complexity is $L_p(1/2, c)$, for some constant c, which is a sub-exponential function. Note that the complexity is given in terms of the size of the smallest unknown prime factor p. If the number is a product of two primes of very unequal size then the elliptic curve method may be the best at finding the factors.
- **Quadratic Sieve:** This is probably the fastest method for factoring integers that have between 80 and 100 decimal digits. It has complexity $L_N(1/2, 1)$.
- **Number Field Sieve:** This is currently the most successful method for numbers with more than 100 decimal digits. It has factored numbers of size $10^{155} \approx 2^{512}$ and has complexity $L_N(1/3, 1.923)$.

Factoring methods are usually divided into Dark Age methods such as

- Trial division,
- $p - 1$ method,
- $p + 1$ method,
- Pollard rho method,

and modern methods such as

- Continued Fraction Method (CFRAC),
- Quadratic Sieve (QS),
- Elliptic Curve Method (ECM),
- Number Field Sieve (NFS).

We do not have space to discuss all of these in detail so we shall look at a couple of Dark Age methods and explain the main ideas behind some of the modern methods.

2.3.1. Trial Division: The most elementary algorithm is trial division, which we have already met in the context of testing primality. Suppose N is the number we wish to factor; we proceed as described in Algorithm 2.4. A moment's thought reveals that trial division takes time at worst

$$O(\sqrt{N}) = O\left(2^{(\log_2 N)/2}\right).$$

The input size to the algorithm is of size $\log_2 N$, hence this complexity is exponential. But just as in primality testing, we should not ignore trial division. It is usually the method of choice for numbers less than 10^{12}.

Algorithm 2.4: Factoring via trial division

```
for p = 2 to √N do
    e ← 0.
    if (N mod p) = 0 then
        while (N mod p) = 0 do
            e ← e + 1.
            N ← N/p.
        output (p, e).
```

2.3.2. Smooth Numbers: For larger numbers we would like to improve on the trial division algorithm. Almost all other factoring algorithms make use of other auxiliary numbers called smooth numbers. Essentially a smooth number is one which is easy to factor using trial division; the following definition makes this more precise.

Definition 2.8 (Smooth Number)**.** *Let B be an integer. An integer N is called B-smooth if every prime factor p of N is less than B.*

For example

$$N = 2^{78} \cdot 3^{89} \cdot 11^3$$

is 12-smooth. Sometimes we say that the number is just smooth if the bound B is small compared with N. The number of y-smooth numbers which are less than x is given by the function $\psi(x, y)$. This is a rather complicated function which is approximated by

$$\psi(x, y) \approx x\rho(u)$$

where ρ is the Dickman–de Bruijn function and

$$u = \frac{\log x}{\log y}.$$

The Dickman–de Bruijn function ρ is defined as the function which satisfies the following differential-delay equation

$$u \cdot \rho'(u) + \rho(u - 1) = 0,$$

for $u > 1$. In practice we approximate $\rho(u)$ via the expression

$$\rho(u) \approx u^{-u},$$

which holds as $u \to \infty$. This leads to the following result, which is important in analysing advanced factoring algorithms.

Theorem 2.9. *The proportion of integers less than x which are $x^{1/u}$-smooth is asymptotically equal to u^{-u}.*

Now if we set $y = L_N(\alpha, \beta)$ then

$$\begin{aligned} u &= \frac{\log N}{\log y} \\ &= \frac{1}{\beta}\left(\frac{\log N}{\log\log N}\right)^{1-\alpha}. \end{aligned}$$

Hence, we can show

$$\begin{aligned} \frac{1}{N}\psi(N, y) &\approx u^{-u} \\ &= \exp(-u \cdot \log u) \\ &\approx \frac{1}{L_N(1-\alpha, \gamma)}, \end{aligned}$$

for some constant γ.

Suppose we are looking for numbers less than N which are $L_N(\alpha, \beta)$-smooth. The probability that any number less than N is actually $L_N(\alpha, \beta)$-smooth is, as we have seen, given by $1/L_N(1-\alpha, \gamma)$. This explains intuitively why some of the modern method complexity estimates for factoring are around $L_N(0.5, c)$, since to balance the smoothness bound against the probability estimate we take $\alpha = \frac{1}{2}$. The Number Field Sieve only obtains a better complexity estimate by using a more mathematically complex algorithm.

We shall also require, in discussing our next factoring algorithm, the notion of a number being B-power smooth:

Definition 2.10 (Power Smooth)**.** *A number is said to be B-power smooth if every prime power dividing N is less than B.*

For example $N = 2^5 \cdot 3^3$ is 33-power smooth.

2.3.3. Pollard's $P-1$ Method: The most famous name in factoring algorithms in the late twentieth century was John Pollard. Almost all the important advances in factoring were made by him, for example

- The $P-1$ method,
- The Rho-method,
- The Number Field Sieve.

In this section we discuss the $P-1$ method and in a later section we consider the Number Field Sieve method.

Suppose the number we wish to factor is given by $N = p \cdot q$. In addition suppose we know (by some pure guess) an integer B such that $p-1$ is B-power smooth, but that $q-1$ is not B-power smooth. We can then hope that $p-1$ divides $B!$, but $q-1$ is unlikely to divide $B!$.

Suppose that we compute

$$a \leftarrow 2^{B!} \pmod{N}.$$

Imagine that we could compute this modulo p and modulo q, we would then have

$$a = 1 \pmod{p},$$

since

- $p-1$ divides $B!$,
- $a^{p-1} = 1 \pmod{p}$ by Fermat's Little Theorem.

Algorithm 2.5: Pollard's $P-1$ factoring method

$a \leftarrow 2$.
for $j = 2$ **to** B **do**
  $a \leftarrow a^j \pmod N$.
$p \leftarrow \gcd(a-1, N)$.
if $p \neq 1$ and $p \neq N$ **then return** "p is a factor of N".
else return "No Result".

But it is unlikely that we would have $a = 1 \pmod q$. Hence,

- p will divide $a-1$,
- q will not divide $a-1$.

We can then recover p by computing $p = \gcd(a-1, N)$, as in Algorithm 2.5

As an example, suppose we wish to factor $N = 15\,770\,708\,441$. We take $B = 180$ and running the above algorithm we obtain

$$a = 2^{B!} \pmod N = 1\,162\,022\,425.$$

Then we obtain

$$p = \gcd(a-1, N) = 135\,979.$$

To see why this works in this example we see that the prime factorization of N is given by

$$N = 135\,979 \cdot 115\,979$$

and we have

$$p - 1 = 135\,978 - 1 = 2 \cdot 3 \cdot 131 \cdot 173,$$
$$q - 1 = 115\,978 - 1 = 2 \cdot 103 \cdot 563.$$

Hence $p-1$ is indeed B-power smooth, whilst $q-1$ is not B-power smooth.

One can show that the complexity of the $P-1$ method is given by

$$O(B \cdot \log B \cdot (\log N)^2 + (\log N)^3).$$

So if we choose $B = O((\log N)^i)$, for some integer i, then this is a polynomial-time factoring algorithm, but it only works for numbers of a special form.

Due to the $P-1$ method we often see it recommended that RSA primes are chosen to satisfy

$$p - 1 = 2 \cdot p_1 \text{ and } q - 1 = 2 \cdot q_1,$$

where p_1 and q_1 are both primes. In this situation the primes p and q are called safe primes. For a random 1024-bit prime p the probability that $p-1$ is B-power smooth, for a small value of B, is very small. Hence, choosing random 1024-bit primes would in all likelihood render the $P-1$ method useless, and so choosing p to be a safe prime is not really needed.

2.3.4. Difference of Two Squares: A basic trick in factoring algorithms, known for many centuries, is to produce two numbers x and y, of around the same size as N, such that

$$x^2 = y^2 \pmod N.$$

Since then we have

$$x^2 - y^2 = (x-y) \cdot (x+y) = 0 \pmod N.$$

If $N = p \cdot q$ then we have four possible cases

(1) p divides $x-y$ and q divides $x+y$.
(2) p divides $x+y$ and q divides $x-y$.
(3) p and q both divide $x-y$ but neither divides $x+y$.
(4) p and q both divide $x+y$ but neither divides $x-y$.

All these cases can occur with equal probability, namely $\frac{1}{4}$. If we then compute

$$d = \gcd(x - y, N),$$

our previous four cases then divide into the cases

(1) $d = p$.
(2) $d = q$.
(3) $d = N$.
(4) $d = 1$.

Since all these cases occur with equal probability, we see that with probability $\frac{1}{2}$ we will obtain a non-trivial factor of N. The only problem is, how do we find x and y such that $x^2 = y^2 \pmod{N}$?

2.4. Modern Factoring Algorithms

Most modern factoring methods use the following strategy based on the difference-of-two-squares method described at the end of the last section.

- Take a smoothness bound B.
- Compute a *factorbase* F of all prime numbers p less than B.
- Find a large number of values of x and y such that x and y are B-smooth and

$$x = y \pmod{N}.$$

  These are called *relations* on the factorbase.
- Using linear algebra modulo 2, find a combination of the relations to give an X and Y with

$$X^2 = Y^2 \pmod{N}.$$

- Attempt to factor N by computing $\gcd(X - Y, N)$.

The trick in all algorithms of this form is how to find the relations. All the other details of the algorithms are basically the same. Such a strategy can be used to solve discrete logarithm problems as well, which we shall discuss in Chapter 3. In this section, we explain the parts of the modern factoring algorithms which are common and justify why they work.

One way of looking at such algorithms is in the context of computational group theory. The factorbase is essentially a set of generators of the group $(\mathbb{Z}/N\mathbb{Z})^*$, whilst the relations are relations between the generators of this group. Once a sufficiently large number of relations have been found, since the group is a finite abelian group, standard group-theoretic algorithms will compute the group structure and hence the group order. From the group order $\phi(N) = (p-1)(q-1)$, we are able to factor the integer N. These general group-theoretic algorithms could include computing the Smith Normal Form of the associated matrix. Hence, it should not be surprising that linear algebra is used on the relations to factor the integer N.

Combining Relations: The Smith Normal Form algorithm is far too complicated for factoring algorithms where a more elementary approach can be used, still based on linear algebra, as we shall now explain. Suppose we have the relations

$$\begin{aligned} p^2 \cdot q^5 \cdot r^2 &= p^3 \cdot q^4 \cdot r^3 \pmod{N}, \\ p \cdot q^3 \cdot r^5 &= p \cdot q \cdot r^2 \pmod{N}, \\ p^3 \cdot q^5 \cdot r^3 &= p \cdot q^3 \cdot r^2 \pmod{N}, \end{aligned}$$

where p, q and r are primes in our factorbase, $F = \{p, q, r\}$. Dividing one side by the other in each of our relations we obtain

$$\begin{aligned} p^{-1} \cdot q \cdot r^{-1} &= 1 \pmod N, \\ q^2 \cdot r^3 &= 1 \pmod N, \\ p^2 \cdot q^2 \cdot r &= 1 \pmod N. \end{aligned}$$

Multiplying the last two equations together we obtain

$$p^{0+2} \cdot q^{2+2} \cdot r^{3+1} = 1 \pmod N.$$

In other words

$$p^2 \cdot q^4 \cdot r^4 = 1 \pmod N.$$

Hence if $X = p \cdot q^2 \cdot r^2$ and $Y = 1$ then we obtain

$$X^2 = Y^2 \pmod N$$

as required and computing

$$\gcd(X - Y, N)$$

will give us a fifty percent chance of factoring N.

Whilst it was easy to see by inspection in the previous example how to combine the relations to obtain a square, in a real-life example our factorbase could consist of hundreds of thousands of primes and we would have hundreds of thousands of relations. We basically need a technique to automate this process of finding out how to combine relations into squares. This is where linear algebra can come to our aid.

We explain how to automate the process using linear algebra by referring to our previous simple example. Recall that our relations were equivalent to

$$\begin{aligned} p^{-1} \cdot q \cdot r^{-1} &= 1 \pmod N, \\ q^2 \cdot r^3 &= 1 \pmod N, \\ p^2 \cdot q^2 \cdot r &= 1 \pmod N. \end{aligned}$$

To find which equations to multiply together to obtain a square, we take a matrix A with $\#F$ columns and number of rows equal to the number of relations. Each relation is coded into the matrix as a row, modulo two, which in our example becomes

$$A = \begin{pmatrix} -1 & 1 & 1 \\ 0 & 2 & 3 \\ 2 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \pmod 2.$$

We now try to find a (non-zero) binary vector z such that

$$z \cdot A = 0 \pmod 2.$$

In our example we can take

$$z = (0, 1, 1)$$

since

$$\begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \pmod 2.$$

This solution vector $z = (0, 1, 1)$ tells us that multiplying the last two equations together will produce a square modulo N.

Finding the vector z is done using a variant of Gaussian Elimination. Hence in general this means that we require more equations (i.e. relations) than elements in the factorbase. This relation-combining stage of factoring algorithms is usually the hardest part since the matrices involved tend to be rather large. For example using the Number Field Sieve to factor a 100-decimal-digit number may require a matrix of dimension over 100 000. This results in huge memory problems and requires the writing of specialist matrix code and often the use of specialized super computers.

The matrix will have around 500 000 rows and as many columns, for cryptographically interesting numbers. As this is nothing but a matrix modulo 2 each entry could be represented by a single bit. If we used a dense matrix representation then the matrix alone would occupy around 29 gigabytes of storage. Luckily the matrix is very, very sparse and so the storage will not be so large.

As we said above, we can compute the vector z such that $z \cdot A = 0$ using a variant of Gaussian Elimination over $\mathbb{Z}/2\mathbb{Z}$. But standard Gaussian Elimination would start with a sparse matrix and end up with an upper triangular dense matrix, so we would be back with the huge memory problem again. To overcome this problem very advanced matrix algorithms are deployed that try not to alter the matrix at all. We do not discuss these here but refer the interested reader to the book of Lenstra and Lenstra mentioned in the Further Reading section of this chapter. The only thing we have not sketched is how to find the relations, a topic which we shall discuss in the next section.

2.5. Number Field Sieve

The Number Field Sieve is the fastest known factoring algorithm. The basic idea is to factor a number N by finding two integers x and y such that

$$x^2 = y^2 \pmod{N};$$

we then expect (hope) that $\gcd(x-y, N)$ will give us a non-trivial factor of N. To explain the basic method we shall start with the linear sieve and then show how this is generalized to the Number Field Sieve. The linear sieve is not a very good algorithm but it does show the rough method.

2.5.1. The Linear Sieve: We let F denote a set of "small" prime numbers which form the factorbase:

$$F = \{p : p \leq B\}.$$

A number which factorizes with all its factors in F is therefore B-smooth. The idea of the linear sieve is to find many pairs of integers a and λ such that

$$b = a + N \cdot \lambda$$

is B-smooth. If in addition we only select values of a which are "small", then we would expect that a will also be B-smooth and we could write

$$a = \prod_{p \in F} p^{a_p}$$

and

$$b = a + N \cdot \lambda = \prod_{p \in F} p^{b_p}.$$

We would then have a relation in $\mathbb{Z}/N\mathbb{Z}$

$$\prod_{p \in F} p^{a_p} = \prod_{p \in F} p^{b_p} \pmod{N}.$$

So the main question is how do we find such values of a and λ?

(1) Fix a value of λ to consider.
(2) Initialize an array of length $A + 1$ indexed by 0 to A with zeros, for some value of A.
(3) For each prime $p \in F$ add $\log_2 p$ to every array location whose position is congruent to $-\lambda \cdot N \pmod{p}$.

(4) Choose the candidates for a to be the positions of those elements that exceed some threshold bound.

The reasoning behind this method is that a position of the array that has an entry exceeding some bound will have a good chance of being B-smooth, when added to λN, as it is likely to be divisible by many primes in F. This is yet another application of the Sieve of Eratosthenes.

Linear Sieve Example: For example suppose we take $N = 1159$, $F = \{2, 3, 5, 7, 11\}$ and $\lambda = -2$. So we wish to find a smooth value of

$$a - 2N.$$

We initialize the sieving array as follows:

0	1	2	3	4	5	6	7	8	9
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We now take the first prime in F, namely $p = 2$, and we compute $-\lambda \cdot N \pmod p = 0$. So we add $\log_2(2) = 1$ to every array location with index equal to 0 modulo 2. This results in our sieve array becoming:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

We now take the next prime in F, namely $p = 3$, and compute $-\lambda \cdot N \pmod p = 2$. So we add $\log_2(3) = 1.6$ to every array location with index equal to 2 modulo 3. Our sieve array then becomes:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	2.6	0.0	1.0	1.6	1.0	0.0	2.6	0.0

Continuing in this way with $p = 5, 7$ and 11, eventually the sieve array becomes:

0	1	2	3	4	5	6	7	8	9
1.0	2.8	2.6	2.3	1.0	1.6	1.0	0.0	11.2	0.0

Hence, the value $a = 8$ looks like it should correspond to a smooth value, and indeed it does, since we find

$$a - \lambda \cdot N = 8 - 2 \cdot 1159 = -2310 = -2 \cdot 3 \cdot 5 \cdot 7 \cdot 11.$$

So using the linear sieve we obtain a large collection of numbers, a_i and b_i, such that

$$a_i = \prod_{p_j \in F} p_j^{a_{i,j}} = \prod_{p_j \in F} p_j^{b_{i,j}} = b_i \pmod N.$$

We assume that we have at least $|B| + 1$ such relations with which we then form a matrix with the ith row being

$$(a_{i,1}, \ldots, a_{i,t}, b_{i,1}, \ldots, b_{i,t}) \pmod 2.$$

We then find elements of the kernel of this matrix modulo 2. This will tell us how to multiply the a_i and the b_i together to obtain elements x^2 and y^2 such that $x, y \in \mathbb{Z}$ are easily calculated and

$$x^2 = y^2 \pmod N.$$

We can then try to factor N, but if these values of x and y do not provide a factor we just find a new element in the kernel of the matrix and continue.

The basic linear sieve gives a very small yield of relations. There is a variant called the *large prime variation* which relaxes the sieving condition to allow through pairs a and b which are almost B-smooth, bar say a single "large" prime in a and a single "large" prime in b. These large primes then have to be combined in some way so that the linear algebra step can proceed as above. This is done by constructing a graph and using an algorithm which computes a basis for the set of cycles in the graph. The basic idea for the large prime variation originally arose in the context of the quadratic sieve algorithm, but it can be applied to any of the sieving algorithms used in factoring.

It is clear that the sieving could be carried out in parallel, hence the sieving can be parcelled out to lots of slave computers around the world. The slaves then communicate any relations they find to the central master computer which performs the linear algebra step. In such a way the Internet can be turned into a large parallel computer dedicated to factoring numbers. As we have already remarked, the final (linear algebra) step often needs to be performed on specialized equipment with large amounts of disk space and RAM, so this final computation cannot be distributed over the Internet.

2.5.2. Higher-Degree Sieving: The linear sieve is simply not good enough to factor large numbers. Indeed, the linear sieve was never proposed as a real factoring algorithm, but its operation is instructive for other algorithms of this type. The Number Field Sieve (NFS) uses the arithmetic of algebraic number fields to construct the desired relations between the elements of the factor-base. All that changes is the way the relations are found. The linear algebra step, the large prime variations and the slave/master approach all go over to NFS virtually unchanged. We now explain the NFS, but in a much simpler form than is actually used in real life so as to aid the exposition. Those readers who do not know any algebraic number theory may wish to skip this section.

First we construct two monic, irreducible polynomials with integer coefficients f_1 and f_2, of degree d_1 and d_2 respectively, such that there exists an $m \in \mathbb{Z}$ such that

$$f_1(m) = f_2(m) = 0 \pmod{N}.$$

The Number Field Sieve will make use of arithmetic in the number fields K_1 and K_2 given by

$$K_1 = \mathbb{Q}(\theta_1) \text{ and } K_2 = \mathbb{Q}(\theta_2),$$

where θ_1 and θ_2 are defined by $f_1(\theta_1) = f_2(\theta_2) = 0$. We then have two homomorphisms ϕ_1 and ϕ_2 given by

$$\phi_i : \begin{cases} \mathbb{Z}[\theta_i] \longrightarrow \mathbb{Z}/N\mathbb{Z} \\ \theta_i \longmapsto m. \end{cases}$$

We aim to use a sieve, just as in the linear sieve, to find a set

$$S \subset \{(a, b) \in \mathbb{Z}^2 : \gcd(a, b) = 1\}$$

such that

$$\prod_S (a - b \cdot \theta_1) = \beta^2$$

and

$$\prod_S (a - b \cdot \theta_2) = \gamma^2,$$

where $\beta \in K_1$ and $\gamma \in K_2$. If we found two such values of β and γ then we would have

$$\phi_1(\beta)^2 = \phi_2(\gamma)^2 \pmod{N}$$

and we hope

$$\gcd(N, \phi_1(\beta) - \phi_2(\gamma))$$

would be a factor of N.

This leads to three obvious problems, which we address in the following three sub-sections:

- How do we find the set S?
- Given $\beta^2 \in \mathbb{Q}[\theta_1]$, how do we compute β?
- How do we find the polynomials f_1 and f_2 in the first place?

How do we find the set S?: Similar to the linear sieve we can find such a set S using linear algebra provided we can find lots of a and b such that

$$a - b \cdot \theta_1 \text{ and } a - b \cdot \theta_2$$

are both "smooth". But what does it mean for these two objects to be smooth? This is rather complicated, and for the rest of this section we will assume the reader has a basic acquaintance with algebraic number theory. It is here that the theory of algebraic number fields comes in: by generalizing our earlier definition of smooth integers to algebraic integers we obtain the following definition:

Definition 2.11. *An algebraic integer is "smooth" if and only if the ideal it generates is only divisible by "small" prime ideals.*

Define $F_i(X, Y) = Y^{d_i} \cdot f_i(X/Y)$, then

$$N_{\mathbb{Q}(\theta_i)/\mathbb{Q}}(a - b \cdot \theta_i) = F_i(a, b).$$

We define two factorbases, one for each of the polynomials

$$\mathcal{F}_i = \{(p, r) : p \text{ a prime}, r \in \mathbb{Z} \text{ such that } f_i(r) = 0 \pmod p\}.$$

Each element of $\mathcal{F}_i$ corresponds to a degree-one prime ideal of $\mathbb{Z}[\theta_i]$, which is a sub-order of the ring of integers of $\mathcal{O}_{\mathbb{Q}(\theta_i)}$, given by

$$\langle p, \theta_i - r\rangle := p\mathbb{Z}[\theta_i] + (\theta_i - r)\mathbb{Z}[\theta_i].$$

Given values of a and b we can easily determine whether the ideal $\langle a - \theta_i \cdot b\rangle$ "factorizes" over our factorbase. Note factorizes is in quotes as unique factorization of ideals may not hold in $\mathbb{Z}[\theta_i]$, whilst it will hold in $\mathcal{O}_{\mathbb{Q}(\theta_i)}$. It will turn out that this is not really a problem. To see why this is not a problem you should consult the book by Lenstra and Lenstra.

If $\mathbb{Z}[\theta_i] = \mathcal{O}_{\mathbb{Q}(\theta_i)}$ then the following method does indeed give the unique prime ideal factorization of $\langle a - \theta_i \cdot b\rangle$.

- Write
$$F_i(a, b) = \prod_{(p_j, r) \in \mathcal{F}_i} p_j^{s_j^{(i)}}.$$
- We have $(a : b) = (r : 1) \pmod p$, as an element in the projective space of dimension one over $\mathbb{F}_p$ (i.e. $a/b = r \pmod p$), if the ideal corresponding to (p, r) is included in a non-trivial way in the ideal factorization of $a - \theta_i b$.
- We have
$$\langle a - \theta_i \cdot b\rangle = \prod_{(p_j, r) \in \mathcal{F}_i} \langle p_j, \theta_i - r\rangle^{s_j^{(i)}}.$$

This leads to the following algorithm to sieve for values of a and b, such that $\langle a - \theta_i \cdot b\rangle$ is an ideal which factorizes over the factorbase. Just as with the linear sieve, the use of sieving allows us to avoid lots of expensive trial divisions when trying to determine smooth ideals. We end up only performing factorizations where we already know we have a good chance of being successful.

- Fix a.
- Initialize the sieve array for $-B \leq b \leq B$ by
$$S[b] = \log_2(F_1(a, b) \cdot F_2(a, b)).$$
- For every $(p, r) \in \mathcal{F}_i$ subtract $\log_2 p$ from every array element $S[b]$ where b is such that
$$a - r \cdot b = 0 \pmod p.$$
- The values of b we want are the ones such that $S[b]$ lies below some tolerance level.

If the tolerance level is set in a sensible way then we have a good chance that both $F_1(a,b)$ and $F_2(a,b)$ factor over the prime ideals in the factorbase, with the possibility of some large prime ideals creeping in. We keep these factorizations as a relation, just as we did with the linear sieve.

Then, after some linear algebra, we can find a subset S of all the pairs (a,b) we have found such that

$$\prod_{(a,b)\in S} \langle a - b\theta_i \rangle = \text{square of an } \textbf{ideal} \text{ in } \mathbb{Z}[\theta_i].$$

However, this is not good enough. Recall that we want the product $\prod(a - b \cdot \theta_i)$ to be the square of an **element** of $\mathbb{Z}[\theta_i]$. To overcome this problem we need to add information from the "infinite" places. This is done by adding in some quadratic characters, an idea introduced by Adleman. Let q be a rational prime (in neither $\mathcal{F}_1$ nor $\mathcal{F}_2$) such that there is an s_q with $f_i(s_q) = 0 \pmod q$ and $f_i'(s_q) \neq 0 \pmod q$ for either $i = 1$ or $i = 2$. Then our extra condition is that we require

$$\prod_{(a,b)\in S} \left(\frac{a - b \cdot s_q}{q}\right) = 1,$$

where $(\frac{\cdot}{q})$ denotes the Legendre symbol. As the Legendre symbol is multiplicative this gives us an extra condition to put into our matrix. We need to add this condition for a number of primes q, hence we choose a set of such primes q and put the associated characters into our matrix as an extra column of 0s or 1s corresponding to:

$$\text{if } \left(\frac{a - b \cdot s_q}{q}\right) = \begin{cases} 1 & \text{then enter} \quad 0, \\ -1 & \text{then enter} \quad 1. \end{cases}$$

After finding enough relations we hope to be able to find a subset S such that

$$\prod_S (a - b \cdot \theta_1) = \beta^2 \text{ and } \prod_S (a - b \cdot \theta_2) = \gamma^2.$$

How do we take the square roots?: We then need to be able to take the square root of β^2 to recover β, and similarly for γ^2. Each β^2 is given in the form

$$\beta^2 = \sum_{j=0}^{d_1-1} a_j \cdot \theta_1^j$$

where the a_j are huge integers. We want to be able to determine the solutions $b_j \in \mathbb{Z}$ to the equation

$$\left(\sum_{j=0}^{d_1-1} b_j \cdot \theta_1^j\right)^2 = \sum_{j=0}^{d_1-1} a_j \cdot \theta_1^j.$$

One way this is overcome, due to Couveignes, is by computing such a square root modulo a large number of very, very large primes p. We then perform Hensel lifting and Chinese remaindering to hopefully recover our square root. This is the easiest method to understand although more advanced methods are available.

Choosing the initial polynomials: This is the part of the method that is a black art at the moment. We require only the following conditions to be met

$$f_1(m) = f_2(m) = 0 \pmod N.$$

However there are good heuristic reasons why it also might be desirable to construct polynomials with additional properties such as

- The polynomials have small coefficients.

- f_1 and f_2 have "many" real roots. Note, a random polynomial probably would have no real roots on average.
- f_1 and f_2 have "many" roots modulo lots of small prime numbers.
- The Galois groups of f_1 and f_2 are "small".

It is often worth spending a few weeks trying to find a good couple of polynomials before we start to attempt the factorization algorithm proper. There are a number of search strategies used for finding these polynomials. Once a few candidates are found, some experimental sieving is performed to see which appear to be the most successful, in that they yield the most relations. Then, once a decision has been made we can launch the sieving stage "for real".

Example: I am grateful to Richard Pinch for allowing me to include the following example. It is taken from his lecture notes from a course at Cambridge in the mid-1990s. Suppose we wish to factor the number $N = 290^2 + 1 = 84\,101$. We take $f_1(x) = x^2 + 1$ and $f_2(x) = x - 290$ with $m = 290$. Then

$$f_1(m) = f_2(m) = 0 \pmod{N}.$$

On one side we have the order $\mathbb{Z}[i]$ which is the ring of integers of $\mathbb{Q}(i)$ and on the other side we have the order $\mathbb{Z}$. We obtain the following factorizations:

x	y	$N(x - i \cdot y)$	Factors	$x - m \cdot y$	Factors
-38	-1	1445	$5 \cdot 17^2$	252	$2^2 \cdot 3^2 \cdot 7$
-22	-19	845	$5 \cdot 13^2$	5488	$2^4 \cdot 7^3$

We then obtain the two factorizations, which are real factorizations of elements, as $\mathbb{Z}[i]$ is a unique factorization domain,

$$-38 + i = -(2 + i) \cdot (4 - i)^2 \text{ and } -22 + 19 \cdot i = -(2 + i) \cdot (3 - 2 \cdot i)^2.$$

Hence, after a trivial bit of linear algebra, we obtain the following "squares"

$$(-38 + i) \cdot (-22 + 19 \cdot i) = (2 + i)^2 \cdot (3 - 2 \cdot i)^2 \cdot (4 - i)^2 = (31 - 12 \cdot i)^2$$

and

$$(-38 + m) \cdot (-22 + 19 \cdot m) = 2^6 \cdot 3^2 \cdot 7^4 = 1176^2.$$

We then apply the map ϕ_1 to $31 - 12 \cdot i$ to obtain

$$\phi_1(31 - 12 \cdot i) = 31 - 12 \cdot m = -3449.$$

But then we have

$$\begin{aligned}
(-3449)^2 &= \phi_1(31 - 12 \cdot i)^2 \\
&= \phi_1((31 - 12 \cdot i)^2) \\
&= \phi_1((-38 + i) \cdot (-22 + 19 \cdot i)) \\
&= \phi_1(-38 + i) \cdot \phi_1(-22 + 19 \cdot i) \\
&= (-38 + m) \cdot (-22 + 19 \cdot m) \pmod{N} \\
&= 1176^2.
\end{aligned}$$

So we compute

$$\gcd(N, -3449 + 1176) = 2273$$

and

$$\gcd(N, -3449 - 1176) = 37.$$

Hence 37 and 2273 are factors of $N = 84\,101$.

Chapter Summary

- Prime numbers are very common and the probability that a random n-bit number is prime is around $1/n$.
- Numbers can be tested for primality using a probable prime test such as the Fermat or Miller–Rabin algorithms. The Fermat Test has a problem in that certain composite numbers will always pass the Fermat Test, no matter how we choose the possible witnesses.
- If we really need to be certain that a number is prime then there are primality-proving algorithms which run in polynomial time.
- We introduced the problems FACTOR, SQRROOT and RSA, and the relations between them.
- Factoring algorithms are often based on the problem of finding the difference of two squares.
- Modern factoring algorithms run in two stages: In the first stage we collect many relations on a factorbase by using a process called sieving, which can be done using thousands of computers on the Internet. In the second stage these relations are processed using linear algebra on a big central server. The final factorization is obtained by finding a difference of two squares.

Further Reading

The definitive reference work on computational number theory which deals with many algorithms for factoring and primality proving is the book by Cohen. The book by Bach and Shallit also provides a good reference for primality testing. The main book explaining the Number Field Sieve is the book by Lenstra and Lenstra.

E. Bach and J. Shallit. *Algorithmic Number Theory. Volume 1: Efficient Algorithms.* MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer, 1993.

A. Lenstra and H. Lenstra. *The Development of the Number Field Sieve.* Springer, 1993.

CHAPTER 3

Discrete Logarithms

Chapter Goals

- To examine algorithms for solving the discrete logarithm problem.
- To introduce the Pohlig–Hellman algorithm.
- To introduce the Baby-Step/Giant-Step algorithm.
- To explain the methods of Pollard.
- To show how discrete logarithms can be solved in finite fields using algorithms like those used for factoring.
- To describe the known results on the elliptic curve discrete logarithm problem.

3.1. The DLP, DHP and DDH Problems

In Chapter 2 we examined the hard problem of FACTOR. This gave us some (hopefully) one-way functions, namely the RSA function, the squaring function modulo a composite and the function which multiplies two large numbers together. Another important class of problems are those based on the discrete logarithm problem or its variants. Let $(G, \cdot)$ be a finite abelian group of prime order q, such as a subgroup of the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field (see Chapter 4). The discrete logarithm problem, or DLP, in G is: given $g, h \in G$, find an integer $x \in [0, \ldots, q)$ (if it exists) such that

$$g^x = h.$$

We write $x = \text{dlog}_g(h)$. A diagram for the security game for the discrete logarithm problem is given in Figure 3.1, for a group G of prime order q. Just as for our factoring-based games we define an advantage function as the probability that the adversary wins the game in Figure 3.1 for a group G, i.e. $\text{Adv}_G^{\mathsf{DLP}}(A) = \Pr[A \text{ wins the } \mathsf{DLP} \text{ game in the group } G]$.

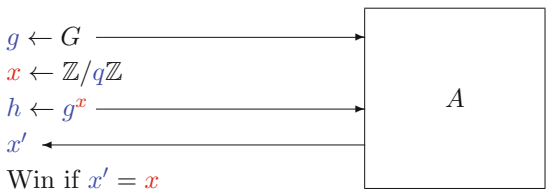


FIGURE 3.1. Security game to define the discrete logarithm problem

For some groups G this problem is easy. For example if we take G to be the integers modulo a number N under addition, then given $g, h \in \mathbb{Z}/N\mathbb{Z}$ we need to solve

$$x \cdot g = h.$$

We have already seen in Chapter 1 that we can easily tell whether such an equation has a solution, and determine its solution when it does, using the extended Euclidean algorithm.

For certain other groups determining discrete logarithms is believed to be hard. For example in the multiplicative group of a finite field the best known algorithm for this task is the Number Field Sieve/Function Field Sieve. The complexity of determining discrete logarithms in this case is given by

$$L_N(1/3, c)$$

for some constant c, depending on the type of the finite field, e.g. whether it is a large prime field or an extension field of small characteristic.

For other groups, such as elliptic curve groups, the discrete logarithm problem is believed to be even harder. The best known algorithm for finding discrete logarithms on a general elliptic curve defined over a finite field $\mathbb{F}_q$ is Pollard's Rho method, a fully exponential algorithm with complexity

$$\sqrt{q} = L_q(1, 1/2).$$

Since determining elliptic curve discrete logarithms is harder than in the case of multiplicative groups of finite fields we are able to use smaller groups. This leads to an advantage in key size. Elliptic curve cryptosystems often have much smaller key sizes (say 256 bits) compared with those based on factoring or discrete logarithms in finite fields (where for both the "equivalent" recommended key size is about 2048 bits).

Later in this chapter we survey the methods known for solving the discrete logarithm problem,

$$h = g^x$$

in various groups G. These algorithms fall into one of two categories: either the algorithms are generic and apply to any finite abelian group or the algorithms are specific to the special group under consideration.

Just as with the FACTOR problem, where we had a number of related problems, with discrete logarithms there are also related problems that we need to discuss. Suppose we are given a finite abelian group $(G, \cdot)$, of prime order q, and $g \in G$. The first of these is the Diffie–Hellman problem.

Definition 3.1 (Computational Diffie–Hellman Problem (DHP))**.** *Given $g \in G$, $a = g^x$ and $b = g^y$, for unknowns x and y chosen at random from $\mathbb{Z}/q\mathbb{Z}$, find c such that $c = g^{x \cdot y}$.*

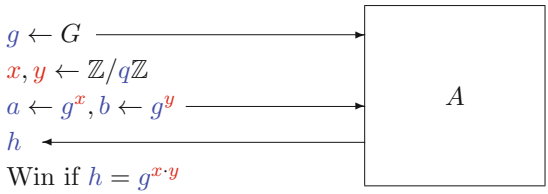


FIGURE 3.2. Security game to define the Computational Diffie–Hellman problem

Diagrammatically we can represent the associated security game as in Figure 3.2, and we define the advantage of the adversary A in the game by

$$\mathrm{Adv}_G^{\mathsf{DHP}}(A) = \Pr[A \text{ wins the } \mathsf{DHP} \text{ game in the group } G].$$

We first show how to reduce solving the Diffie–Hellman problem to the discrete logarithm problem. But before doing so we note that in some groups there is a more complicated argument to show that the DHP is in fact equivalent to the DLP. This is done by producing a reduction in the other direction for the specific groups in question.

Lemma 3.2. *In an arbitrary finite abelian group G the DHP is no harder than the DLP. In particular for all algorithms A there is an algorithm B such that*

$$\mathrm{Adv}_G^{\mathsf{DLP}}(A) = \mathrm{Adv}_G^{\mathsf{DHP}}(B).$$

PROOF. Suppose we have an oracle/algorithm A which will solve the DLP, i.e. on input of $h = g^x$ it will return x. To solve the DHP on input of $a = g^x$ and $b = g^y$ we compute

(1) $z \leftarrow A(a)$.
(2) $c \leftarrow b^z$.
(3) Output c.

The above reduction clearly runs in polynomial time and will compute the true solution to the DHP, assuming algorithm A returns the correct value, i.e. $z = x$. Hence, the DHP is no harder than the DLP. □

There is a decisional version of the DHP problem, just like there is a decisional version QUADRES of the SQRROOT problem.

Definition 3.3 (Decision Diffie–Hellman problem (DDH))**.** *The adversary is given $g \in G$, $a = g^x$, $b = g^y$, and $c = g^z$, for unknowns x, y and z. The value z is chosen by the challenger to be equal to $x \cdot y$ with probability $1/2$, otherwise it is chosen at random. The goal of the adversary is to determine which case he thinks the challenger picked, i.e. he has to determine whether $z = x \cdot y$.*

Diagrammatically we can represent the associated security game as in Figure 3.3. But when defining the advantage for the DDH problem we need to be a bit careful, as the adversary can always win with probability one half, by just guessing the bit b at random. This is exactly the same situation as we had when looking at the QUADRES problem in Chapter 2. We hence define the advantage by

$$\mathrm{Adv}_G^{\mathsf{DDH}}(A) = 2 \cdot \left| \Pr[A \text{ wins the } \mathsf{DDH} \text{ game in } G] - \frac{1}{2} \right|.$$

Notice that with this definition, just as with the advantage in the QUADRES game, if the adversary just guesses the bit with probability $1/2$ then its advantage is zero as we would expect, since $2 \cdot |1/2 - 1/2| = 0$. If however the adversary is always right, or indeed always wrong, then the advantage is one, since $2 \cdot |1 - 1/2| = 2 \cdot |0 - 1/2| = 1$. Thus, the advantage is normalized to lie between zero and one, with one being always successful and zero being no better than random. Just like Lemma 2.3 we have that another way to write down the advantage is

$$\mathrm{Adv}_G^{\mathsf{DDH}}(A) = \left| \Pr[b' = 1 | b = 1] - \Pr[b' = 1 | b = 0] \right|.$$

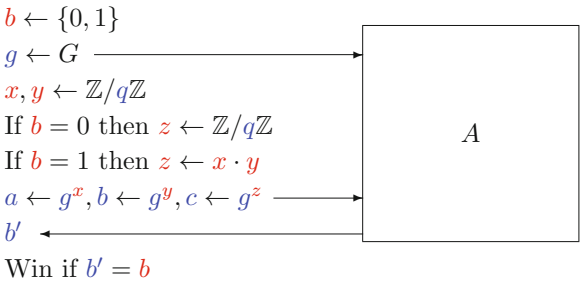

$b \leftarrow \{0,1\}$
$g \leftarrow G$
$x, y \leftarrow \mathbb{Z}/q\mathbb{Z}$
If $b = 0$ then $z \leftarrow \mathbb{Z}/q\mathbb{Z}$
If $b = 1$ then $z \leftarrow x \cdot y$
$a \leftarrow g^x, b \leftarrow g^y, c \leftarrow g^z$
b'
Win if $b' = b$
A

FIGURE 3.3. Security game to define the Decision Diffie–Hellman problem

We now show how to reduce the solution of the Decision Diffie–Hellman problem to the Computational Diffie–Hellman problem, and hence using our previous argument to the discrete logarithm problem.

Lemma 3.4. *In an arbitrary finite abelian group G the DDH is no harder than the DHP. In particular for all algorithms A there is an algorithm B such that*

$$\mathrm{Adv}_G^{\mathsf{DHP}}(A) = \mathrm{Adv}_G^{\mathsf{DDH}}(B).$$

Proof. Suppose we have an oracle A which on input of g^x and g^y computes the value of $g^{x\cdot y}$. To solve the DDH on input of $a = g^x, b = g^y$ and $c = g^z$ we compute

(1) $d \leftarrow A(a, b)$.
(2) If $d = c$ output one.
(3) Else output zero.

Again the reduction clearly runs in polynomial time, and assuming the output of the oracle is correct then the above reduction will solve the DDH. □

So the Decision Diffie–Hellman problem is no harder than the Computational Diffie–Hellman problem. There are, however, some groups[1] in which we can solve the DDH in polynomial time but the fastest known algorithm to solve the DHP takes sub-exponential time. Hence, of our three discrete-logarithm-based problems, the easiest is DDH, then comes DHP and finally the hardest problem is DLP.

3.2. Pohlig–Hellman

The first observation to make is that the discrete logarithm problem in a group G is only as hard as the discrete logarithm problem in the largest subgroup of prime order in G. This observation is due to Pohlig and Hellman, and it applies in an arbitrary finite abelian group. To explain the Pohlig–Hellman algorithm, suppose we have a finite cyclic abelian group $G = \langle g \rangle$ whose order is given by

$$N = \#G = \prod_{i=1}^{t} p_i^{e_i}.$$

Now suppose we are given $h \in \langle g \rangle$, so there exists an integer x such that

$$h = g^x.$$

Our aim is to find x by first finding it modulo $p_i^{e_i}$ and then using the Chinese Remainder Theorem to recover it modulo N.

From basic group theory we know that there is a group isomorphism

$$\phi : G \longrightarrow C_{p_1^{e_1}} \times \cdots \times C_{p_t^{e_t}},$$

where C_{p^e} is a cyclic group of prime power order p^e. The projection of ϕ on the component C_{p^e} is given by

$$\phi_p : \begin{cases} G \longrightarrow C_{p^e} \\ f \longmapsto f^{N/p^e}. \end{cases}$$

The map ϕ_p is a group homomorphism, so if we have $h = g^x$ in G then we will have $\phi_p(h) = \phi_p(g)^x$ in C_{p^e}. But the discrete logarithm in C_{p^e} is only determined modulo p^e. So if we could solve the discrete logarithm problem in C_{p^e}, then we would determine x modulo p^e. Doing this for all primes p dividing N would allow us to solve for x using the Chinese Remainder Theorem. In summary suppose we have some oracle $O(g, h, p, e)$ which for $g, h \in C_{p^e}$ will output the discrete logarithm of h with respect to g. We can then solve for x using Algorithm 3.1.

[1] For example supersingular elliptic curves.

Algorithm 3.1: Algorithm to solve the DLP in a group of order N, given an oracle for DLP for prime power divisors of N

$S \leftarrow \{\}$.
for all primes p dividing N **do**
  Compute the largest e such that $T = p^e$ divides N.
  $g_1 \leftarrow g^{N/T}$.
  $h_1 \leftarrow h^{N/T}$.
  $z \leftarrow O(g_1, h_1, p, e)$.
  $S \leftarrow S \cup \{(z, T)\}$.
$x \leftarrow$CRT(S)

The only problem is that we have not shown how to solve the discrete logarithm problem in C_{p^e}. We shall now show how this is done, by reducing to solving e discrete logarithm problems in the group C_p. Suppose $g, h \in C_{p^e}$ and there is an x such that

$$h = g^x.$$

Clearly x is only defined modulo p^e and we can write

$$x = x_0 + x_1 \cdot p + \cdots + x_{e-1} \cdot p^{e-1}.$$

We find $x_0, x_1, \ldots$ in turn, using the following inductive procedure. Suppose we know x', the value of x modulo p^t, i.e.

$$x' = x_0 + \cdots + x_{t-1} \cdot p^{t-1}.$$

We now wish to determine x_t and so compute x modulo p^{t+1}. We write

$$x = x' + p^t \cdot y,$$

so we have that

$$h = g^{x'} \cdot (g^{p^t})^y.$$

Hence, if we set

$$h_1 = h \cdot g^{-x'} \text{ and } g_1 = g^{p^t},$$

then

$$h_1 = {g_1}^y.$$

Now g_1 is an element of order p^{e-t}, so to obtain an element of order p and hence a discrete logarithm problem in C_p, we need to raise the above equation to the power $s = p^{e-t-1}$. So, setting

$$h_2 = h_1^s \text{ and } g_2 = g_1^s,$$

we obtain the discrete logarithm problem in C_p given by

$$h_2 = {g_2}^{x_t}.$$

So assuming we can solve discrete logarithms in C_p we can find x_t and so find x.

Pohlig–Hellman Example: We now illustrate this approach, assuming we can find discrete logarithms in cyclic groups of prime order. We leave to the next two sections how to do this; for now we assume that it is possible. As an example of the Pohlig–Hellman algorithm, consider the multiplicative group of the finite field $\mathbb{F}_{397}$. This group has order

$$N = 396 = 2^2 \cdot 3^2 \cdot 11$$

and a generator of $\mathbb{F}_{397}^*$ is given by

$$g = 5.$$

We wish to solve the discrete logarithm problem given by

$$h = 208 = 5^x \pmod{397}.$$

We first reduce to the three subgroups of prime power order, by raising the above equation to the power $396/p^e$, for each maximal prime power p^e which divides the order of the group 396. Hence, we obtain the three discrete logarithm problems

$$334 = h^{396/4} = g^{396/4 x_4} = 334^{x_4} \pmod{397},$$
$$286 = h^{396/9} = g^{396/9 x_9} = 79^{x_9} \pmod{397},$$
$$273 = h^{396/11} = g^{396/11 x_{11}} = 290^{x_{11}} \pmod{397}.$$

The value of x_4 is the value of x modulo 4, the value of x_9 is the value of x modulo 9 whilst the value of x_{11} is the value of x modulo 11. Clearly if we can determine these three values then we can determine x modulo 396.

Determining x_4: By inspection we see that $x_4 = 1$, but let us labour the point and show how the above algorithm will determine this for us. We write

$$x_4 = x_{4,0} + 2 \cdot x_{4,1},$$

where $x_{4,0}, x_{4,1} \in \{0, 1\}$. Recall that we wish to solve

$$h_1 = 334 = 334^{x_4} = {g_1}^{x_4}.$$

We set $h_2 = h_1^2$ and $g_2 = g_1^2$ and solve the discrete logarithm problem

$$h_2 = {g_2}^{x_{4,0}}$$

in the cyclic group of order two. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{4,0} = 1$. So we now have

$$\frac{h_1}{g_1} = {g_2}^{x_{4,1}} \pmod{397}.$$

Hence we have $1 = 396^{x_{4,1}}$, which is another discrete logarithm in the cyclic group of order two. We find $x_{4,1} = 0$ and, as expected,

$$x_4 = x_{4,0} + 2 \cdot x_{4,1} = 1 + 2 \cdot 0 = 1.$$

Determining x_9: We write

$$x_9 = x_{9,0} + 3 \cdot x_{9,1},$$

where $x_{9,0}, x_{9,1} \in \{0, 1, 2\}$. Recall that we wish to solve

$$h_1 = 286 = 79^{x_9} = {g_1}^{x_9}.$$

We set $h_2 = h_1^3$ and $g_2 = g_1^3$ and solve the discrete logarithm problem

$$h_2 = 34 = {g_2}^{x_{9,0}} = 362^{x_{9,0}}$$

in the cyclic group of order three. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{9,0} = 2$. So we now have

$$\frac{h_1}{g_1^2} = {g_2}^{x_{9,1}} \pmod{397}.$$

Hence we have $1 = 362^{x_{9,1}}$, which is another discrete logarithm in the cyclic group of order three. We find $x_{9,1} = 0$ and so conclude that

$$x_9 = x_{9,0} + 3 \cdot x_{9,1} = 2 + 3 \cdot 0 = 2.$$

Determining x_{11}: We are already in a cyclic group of prime order, so applying our oracle to the discrete logarithm problem

$$273 = 290^{x_{11}} \pmod{397},$$

we find that $x_{11} = 6$.

Summary: So we have determined that if

$$208 = 5^x \pmod{397},$$

then x is given by

$$\begin{aligned} x &= 1 \pmod 4, \\ x &= 2 \pmod 9, \\ x &= 6 \pmod{11}. \end{aligned}$$

If we apply the Chinese Remainder Theorem to this set of three simultaneous equations, then we obtain that the solution to our discrete logarithm problem is given by $x = 281$.

3.3. Baby-Step/Giant-Step Method

In our above discussion of the Pohlig–Hellman algorithm we assumed we had an oracle to solve the discrete logarithm problem in cyclic groups of prime order. We shall now describe a general method to solve such problems due to Shanks called the Baby-Step/Giant-Step method. We stress that this is a generic method which applies to any cyclic finite abelian group.

Since the intermediate steps in the Pohlig–Hellman algorithm are quite simple, the difficulty of solving a general discrete logarithm problem will be dominated by the time required to solve the discrete logarithm problem in the cyclic subgroups of prime order. Hence, for generic groups the complexity of the Baby-Step/Giant-Step method will dominate the overall complexity of any algorithm. Indeed, one can show that the following method is the best possible method, time-wise, for solving the discrete logarithm problem in an arbitrary group. Of course in any actual group there may be a special purpose algorithm which works faster, but in general the following is provably the best one can do.

We fix notation as follows: We have a public cyclic group $G = \langle g \rangle$, which we can now assume to have prime order p. We are also given an $h \in G$ and are asked to find the value of x modulo p such that

$$h = g^x.$$

We assume there is some fixed encoding of the elements of G, so in particular it is easy to store, sort and search a list of elements of G.

The idea behind the Baby-Step/Giant-Step method is a standard divide-and-conquer approach found in many areas of computer science. We write

$$x = x_0 + x_1 \cdot \lceil \sqrt{p} \rceil.$$

Now, since $0 \leq x \leq p$, we have that $0 \leq x_0, x_1 < \lceil \sqrt{p} \rceil$. We first compute the Baby-Steps

$$g_i \leftarrow g^i \text{ for } 0 \leq i < \lceil \sqrt{p} \rceil.$$

The pairs

$$(g_i, i)$$

are stored in a table so that one can easily search for items indexed by the first entry in the pair. This can be accomplished by sorting the table on the first entry, or more efficiently by the use of hash tables. To compute and store the Baby-Steps clearly requires

$$O(\lceil \sqrt{p} \rceil)$$

time and a similar amount of storage.

We now compute the Giant-Steps $h_j \leftarrow h \cdot g^{-j \cdot \lceil \sqrt{p} \rceil}$, for $0 \leq j < \lceil \sqrt{p} \rceil$, and try to find a match in the table of Baby-Steps, i.e. we try to find a value g_i such that $g_i = h_j$. If such a match occurs we have

$$x_0 = i \text{ and } x_1 = j,$$

since, if $g_i = h_j$, we have

$$g^i = h \cdot g^{-j \cdot \lceil \sqrt{p} \rceil},$$

i.e.

$$g^{i+j \cdot \lceil \sqrt{p} \rceil} = h.$$

Notice that the time to compute the Giant-Steps is at most

$$O(\lceil \sqrt{p} \rceil).$$

Hence, the overall time and space complexity of the Baby-Step/Giant-Step method is

$$O(\sqrt{p}).$$

This means, combining with the Pohlig–Hellman algorithm, that if we wish a discrete logarithm problem in a group G to be as difficult as a work effort of 2^{128} operations, then we need the group G to have a prime order subgroup of size larger than 2^{256}.

Baby-Step/Giant-Step Example: As an example we take the subgroup of order 101 in the multiplicative group of the finite field $\mathbb{F}_{607}$, generated by $g = 64$. Suppose we are given the discrete logarithm problem

$$h = 182 = 64^x \pmod{607}.$$

We first compute the Baby-Steps

$$g_i = 64^i \pmod{607} \text{ for } 0 \leq i < \lceil \sqrt{101} \rceil = 11.$$

We compute

i	$64^i \pmod{607}$	i	$64^i \pmod{607}$
0	1	6	330
1	64	7	482
2	454	8	498
3	527	9	308
4	**343**	10	288
5	100		

Now we compute the Giant-Steps,

$$h_j = 182 \cdot 64^{-11 \cdot j} \pmod{607} \text{ for } 0 \leq j < 11,$$

and check when we obtain a Giant-Step which occurs in our table of Baby-Steps:

j	$182 \cdot 64^{-11 \cdot j} \pmod{607}$	j	$182 \cdot 64^{-11 \cdot j} \pmod{607}$
0	182	6	60
1	143	7	394
2	69	8	483
3	271	9	76
4	**343**	10	580
5	573		

So we obtain a match when $i = 4$ and $j = 4$, which means that

$$x = 4 + 11 \cdot 4 = 48,$$

which we can verify to be the correct answer to the earlier discrete logarithm problem by computing $64^{48} \pmod{607} = 182$.

3.4. Pollard-Type Methods

The trouble with the Baby-Step/Giant-Step method is that, although its run time is bounded by $O(\sqrt{p})$, it required $O(\sqrt{p})$ space. In practice this space requirement is more of a hindrance than the time requirement. Hence, one could ask whether one could trade the large space requirement for a smaller space requirement, but still obtain a time complexity of $O(\sqrt{p})$? Well we can, but we will now obtain only an expected running time rather than an absolute bound on the running time; thus technically we obtain a Las Vegas-style algorithm as opposed to a deterministic one. There are a number of algorithms which achieve this reduced space requirement all of which are due to ideas of Pollard.

3.4.1. Pollard's Rho Algorithm: Suppose $f : S \to S$ is a random mapping between a set S and itself, where the size of S is n. Now pick a random value $x_0 \in S$ and compute

$$x_{i+1} \leftarrow f(x_i) \text{ for } i \geq 0.$$

We consider the values $x_0, x_1, x_2, \ldots$ as a deterministic random walk. By this statement we mean that each step $x_{i+1} = f(x_i)$ of the walk is a deterministic function of the current position x_i, but we are assuming that the sequence $x_0, x_1, x_2, \ldots$ *behaves* as a random sequence would. Another name for a deterministic random walk is a pseudo-random walk.

The goal of many of Pollard's algorithms is to find a collision in a random mapping like the one above, where a collision is finding a pair of values x_i and x_j with $i \neq j$ such that

$$x_i = x_j.$$

From the birthday paradox from Section 1.4.2, we obtain a collision after an expected number of

$$\sqrt{\pi \cdot n/2}$$

iterations of the map f. Hence, finding a collision using the birthday paradox in a naive way would require $O(\sqrt{n})$ time and $O(\sqrt{n})$ memory. But this large memory requirement is exactly the problem with the Baby-Step/Giant-Step method we were trying to avoid.

But, since S if finite, we must eventually obtain $x_i = x_j$ for *some* values of i and j, and so

$$x_{i+1} = f(x_i) = f(x_j) = x_{j+1}.$$

Hence, the sequence $x_0, x_1, x_2, \ldots$, will eventually become cyclic. If we "draw" such a sequence then it looks like the Greek letter rho, ρ. In other words there is a cyclic part and an initial tail. It can be shown, using much the same reasoning as for the birthday bound above, that for a random mapping, the tail has expected length $\sqrt{\pi \cdot n/8}$, whilst the cycle also has expected length $\sqrt{\pi \cdot n/8}$. It is this observation which will allow us to reduce the memory requirement to constant space.

To find a collision and make use of the rho shape of the random walk, we use a technique called Floyd's cycle-finding algorithm: Given (x_1, x_2) we compute (x_2, x_4) and then (x_3, x_6) and so on, i.e. given the pair (x_i, x_{2i}) we compute

$$(x_{i+1}, x_{2i+2}) = (f(x_i), f(f(x_{2i}))).$$

We stop when we find

$$x_m = x_{2m}.$$

If the tail of the sequence $x_0, x_1, x_2, \ldots$ has length λ and the cycle has length μ then it can be shown that we obtain such a value of m when

$$m = \mu \cdot (1 + \lfloor \lambda/\mu \rfloor).$$

Since $\lambda < m \leq \lambda + \mu$ we see that

$$m = O(\sqrt{n}),$$

and this will be an accurate complexity estimate if the mapping f behaves suitably like a random function. Hence, we can detect a collision with virtually no storage.

This is all very well, but we have not shown how to relate this to the discrete logarithm problem. Let G denote a group of order n and let the discrete logarithm problem be given by

$$h = g^x.$$

We partition the group into three sets S_1, S_2, S_3, where we assume $1 \notin S_2$, and then define the following random walk on the group G,

$$x_{i+1} \leftarrow f(x_i) = \begin{cases} h \cdot x_i & x_i \in S_1, \\ x_i^2 & x_i \in S_2, \\ g \cdot x_i & x_i \in S_3. \end{cases}$$

The condition that $1 \notin S_2$ is to ensure that the function f has no stationary points. In practice we actually keep track of three pieces of information

$$(x_i, a_i, b_i) \in G \times \mathbb{Z} \times \mathbb{Z}$$

where

$$a_{i+1} \leftarrow \begin{cases} a_i & x_i \in S_1, \\ 2 \cdot a_i \pmod{n} & x_i \in S_2, \\ a_i + 1 \pmod{n} & x_i \in S_3, \end{cases}$$

and

$$b_{i+1} \leftarrow \begin{cases} b_i + 1 \pmod{n} & x_i \in S_1, \\ 2 \cdot b_i \pmod{n} & x_i \in S_2, \\ b_i & x_i \in S_3. \end{cases}$$

If we start with the triple

$$(x_0, a_0, b_0) = (1, 0, 0)$$

then we have, for all i,

$$\log_g(x_i) = a_i + b_i \cdot \log_g(h) = a_i + b_i \cdot x.$$

Applying Floyd's cycle-finding algorithm we obtain a collision, and so find a value of m such that

$$x_m = x_{2m}.$$

This leads us to deduce the following equality of discrete logarithms

$$\begin{aligned} a_m + b_m \cdot x &= a_m + b_m \cdot \log_g(h) \\ &= \log_g(x_m) \\ &= \log_g(x_{2m}) \\ &= a_{2m} + b_{2m} \cdot \log_g(h) \\ &= a_{2m} + b_{2m} \cdot x. \end{aligned}$$

Rearranging we see that

$$(b_m - b_{2m}) \cdot x = a_{2m} - a_m,$$

and so, if $b_m \neq b_{2m}$, we obtain

$$x = \frac{a_{2m} - a_m}{b_m - b_{2m}} \pmod{n}.$$

The probability that we have $b_m = b_{2m}$ is small enough to be ignored for large n. Thus the above algorithm will find the discrete logarithm in expected time $O(\sqrt{n})$.

Pollard's Rho Example: As an example consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$ and the discrete logarithm problem

$$h = 122 = 64^x.$$

We define the sets S_1, S_2, S_3 as follows:

$$\begin{aligned} S_1 &= \{x \in \mathbb{F}_{607}^* : x \leq 201\}, \\ S_2 &= \{x \in \mathbb{F}_{607}^* : 202 \leq x \leq 403\}, \\ S_3 &= \{x \in \mathbb{F}_{607}^* : 404 \leq x \leq 606\}. \end{aligned}$$

Applying Pollard's Rho method we obtain the following data

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
0	1	0	0	1	0	0
1	122	0	1	316	0	2
2	316	0	2	172	0	8
3	308	0	4	137	0	18
4	172	0	8	7	0	38
5	346	0	9	309	0	78
6	137	0	18	352	0	56
7	325	0	19	167	0	12
8	7	0	38	498	0	26
9	247	0	39	172	2	52
10	309	0	78	137	4	5
11	182	0	55	7	8	12
12	352	0	56	309	16	26
13	76	0	11	352	32	53
14	**167**	0	12	**167**	64	6

So we obtain a collision, using Floyd's cycle-finding algorithm, when $m = 14$. We see that

$$g^0 \cdot h^{12} = g^{64} \cdot h^6$$

which implies

$$12 \cdot x = 64 + 6 \cdot x \pmod{101}.$$

Consequently

$$x = \frac{64}{12 - 6} \pmod{101} = 78.$$

3.4.2. Pollard's Lambda Method: Pollard's Lambda method is like the Rho method, in that we use a deterministic random walk and a small amount of storage to solve the discrete logarithm problem. However, the Lambda method is particularly tuned to the situation where we know that the discrete logarithm lies in a certain interval

$$x \in [a, \ldots, b].$$

In the Rho method we used one random walk, which turned into the shape of the Greek letter ρ, whilst in the Lambda method we use two walks which end up in the shape of the Greek letter lambda, i.e. λ, hence giving the method its name. Another name for this method is Pollard's Kangaroo method as it was originally described with the two walks being performed by kangaroos.

Let $w = b - a$ denote the length of the interval in which the discrete logarithm x is known to lie. We define a set

$$S = \{s_0, \ldots, s_{k-1}\}$$

of integers in non-decreasing order. The mean m of the set should be around $N = \lfloor \sqrt{w} \rfloor$. It is common to choose

$$s_i = 2^i \text{ for } 0 \leq i < k,$$

which implies that the mean of the set is

$$\frac{2^k}{k},$$

and so we choose

$$k \approx \frac{1}{2} \cdot \log_2(w).$$

We divide the group up into k sets S_i, for $i = 0, \ldots, k-1$, and define the following deterministic random walk:

$$x_{i+1} = x_i \cdot g^{s_j} \text{ if } x_i \in S_j.$$

We first compute the deterministic random walk starting from the end of the interval $g_0 = g^b$ by setting

$$g_{i+1} = g_i \cdot g^{s_j} \text{ if } g_i \in S_j,$$

for $i = 1, \ldots, N = \lfloor \sqrt{w} \rfloor$. We also set $c_0 = b$ and $c_{i+1} = c_i + s_j \pmod{q}$. We store g_N and notice that we have computed the discrete logarithm of g_N with respect to g,

$$c_N = \log_g(g_N).$$

We now compute our second deterministic random walk, starting from the point in the interval corresponding to the unknown x; we set $h_0 = h = g^x$ and compute

$$h_{i+1} = h_i \cdot g^{s_j} \text{ if } h_i \in S_j.$$

We also set $d_0 = 0$ and $d_{i+1} = d_i + s_j \pmod{q}$. Notice that we have

$$\log_g(h_i) = x + d_i.$$

If the path of the h_i ever meets that of the path of the g_i then the h_i will carry on the path of the g_i, and so eventually reach the point g_N. Thus, we are able to find a value M where h_M equals our stored point g_N. We then have

$$c_N = \log_g(g_N) = \log_g(h_M) = x + d_M,$$

and so the solution to our discrete logarithm problem is given by

$$x = c_N - d_M \pmod{q}.$$

If we do not get a collision then we can increase N and continue both walks in a similar manner until a collision does occur.

The expected running time of this method is $\sqrt{w}$ and again the storage can be seen to be constant. The Lambda method can be used when the discrete logarithm is only known to lie in

the full interval $[0, \ldots, q-1]$. But in this situation, whilst the asymptotic complexity is the same as the Rho method, the Rho method is better due to the implied constants.

Pollard's Lambda Example: As an example we again consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$, but now we look at the discrete logarithm problem

$$h = 524 = 64^x.$$

We are given that the discrete logarithm x lies in the interval $[60, \ldots, 80]$. As our set of multipliers s_i we take $s_i = 2^i$ for $i = 0, 1, 2, 3$. The subsets $S_0, \ldots, S_3$ of G we define by

$$S_i = \{g \in G : g \pmod 4 = i\}.$$

We first compute the deterministic random walk g_i and the discrete logarithms $c_i = \log_g(g_i)$, for $i = 0, \ldots, N = \lfloor \sqrt{80 - 40} \rfloor = 4$.

i	g_i	c_i
0	151	80
1	537	88
2	391	90
3	478	98
4	64	1

Now we compute the second deterministic random walk

i	h_i	$d_i = \log_g(h_i) - x$
0	524	0
1	151	1
2	537	9
3	391	11
4	478	19
5	64	23

Hence, we obtain the collision $h_5 = g_4$ and so

$$x = 1 - 23 \pmod{101} = 79.$$

Note that examining the above tables we see that we had earlier collisions between our two walks. However, we are unable to use these since we do not store g_0, g_1, g_2 or g_3. We have only stored the value of g_4.

3.4.3. Parallel Pollard's Rho: In real life when we use random-walk-based techniques to solve discrete logarithm problems we use a parallel version, to exploit the computing resources of a number of sites across the Internet. Suppose we are given the discrete logarithm problem

$$h = g^x$$

in a group G of prime order q. We first decide on an easily computable function

$$H : G \longrightarrow \{1, \ldots, k\},$$

where k is usually around 20. Then we define a set of multipliers m_i. These are produced by generating random integers $a_i, b_i \in [0, \ldots, q-1]$ and then setting

$$m_i = g^{a_i} \cdot h^{b_i}.$$

To start a deterministic random walk we pick random $s_0, t_0 \in [0, \ldots, q-1]$ and compute

$$g_0 = g^{s_0} \cdot h^{t_0},$$

the deterministic random walk is then defined on the triples (g_i, s_i, t_i) where

$$\begin{aligned} g_{i+1} &= g_i \cdot m_{H(g_i)}, \\ s_{i+1} &= s_i + a_{H(g_i)} \pmod{q}, \\ t_{i+1} &= t_i + b_{H(g_i)} \pmod{q}. \end{aligned}$$

Hence, for every g_i we record the values of s_i and t_i such that

$$g_i = g^{s_i} \cdot h^{t_i}.$$

Suppose we have m processors: each processor starts a different deterministic random walk from a different starting position using the same algorithm to determine the next element in the walk. When another processor, or even the same processor, meets an element of the group that has been seen before then we obtain an equation

$$g^{s_i} \cdot h^{t_i} = g^{s'_j} \cdot h^{t'_j}$$

which we can solve for the discrete logarithm x. Hence, we expect that after $O(\sqrt{(\pi \cdot q)/(2 \cdot m^2)})$ iterations of these parallel walks we will find a collision and so solve the discrete logarithm problem.

However, as described this means that each processor needs to return every element in its computed deterministic random walk to a central server which then stores all the computed elements. This is highly inefficient as the storage requirements will be very large, namely $O(\sqrt{(\pi \cdot q)/2})$. We can reduce the storage to any required value as follows: We first define a function d on the group

$$d : G \longrightarrow \{0, 1\}$$

such that $d(g) = 1$ around $1/2^t$ of the time for any $g \in G$. The function d is often defined by returning $d(g) = 1$ if a certain subset of t of the bits representing the group element $g \in G$ are set to zero for example. The elements in G for which $d(g) = 1$ will be called distinguished.

It is only the distinguished group elements that are now transmitted back to the central server. This means that we expect the deterministic random walks to need to continue another 2^t steps before a collision is detected between two deterministic random walks. Hence, the computing time now becomes

$$O\left(\sqrt{(\pi \cdot q)/(2 \cdot m^2)}/ + 2^t\right),$$

whilst the storage becomes

$$O\left(\sqrt{(\pi \cdot q)/2^{2 \cdot t+1}}\right).$$

This allows the storage to be reduced to any manageable amount, at the expense of a little extra computation. We do not give an example, since the method only really becomes useful as q becomes large (say $q > 2^{20}$).

3.5. Sub-exponential Methods for Finite Fields

There is a close relationship between the sub-exponential methods for factoring and the sub-exponential methods for solving the discrete logarithm problem in finite fields. We shall only consider the case of prime fields $\mathbb{F}_p$ but similar considerations apply to finite fields of small characteristic; here we use a special algorithm called the Function Field Sieve. The sub-exponential algorithms for finite fields are often referred to as index-calculus algorithms, as an index is an old name for a discrete logarithm.

We assume we are given $g, h \in \mathbb{F}_p^*$ such that

$$h = g^x.$$

We choose a factorbase F of elements, usually small prime numbers, and then, using one of the sieving strategies used for factoring, we obtain a large number of relations of the form

$$\prod_{p_i \in F} p_i^{e_i} = 1 \pmod{p}.$$

These relations translate into the following equations for discrete logarithms,

$$\sum_{p_i \in F} e_i \cdot \log_g(p_i) = 0 \pmod{p-1}.$$

Once enough equations like the one above have been found we can solve for the discrete logarithm of every element in the factorbase, i.e. we can determine

$$x_i = \log_g(p_i).$$

The value of x_i is sometimes called the index of p_i with respect to g. This calculation is performed using linear algebra modulo $p-1$, which is more complicated than the linear algebra modulo two performed in factoring algorithms. However similar tricks, to those deployed in the linear algebra stage of factoring algorithms can be deployed to keep storage requirements down to manageable levels.

This linear algebra calculation only needs to be done once for each generator g, and the results can then be used for many values of h. When we wish to solve a particular discrete logarithm problem $h = g^x$, we use a sieving technique, or simple trial and error, to write

$$h = \prod_{p_i \in F} p_i^{h_i} \pmod{p},$$

e.g. we could compute

$$T = h \cdot \prod_{p_i \in F} p_i^{f_i} \pmod{p}$$

and see whether it factors in the form

$$T = \prod_{p_i \in F} p_i^{g_i}.$$

If it does then we have

$$h = \prod_{p_i \in F} p_i^{g_i - f_i} \pmod{p}.$$

We can then compute the discrete logarithm x from

$$\begin{aligned} x = \log_g(h) &= \log_g\left(\prod_{p_i \in F} p_i^{h_i}\right) \\ &= \sum_{p_i \in F} h_i \cdot \log_g(p_i) \pmod{p-1} \\ &= \sum_{p_i \in F} h_i \cdot x_i \pmod{p-1}. \end{aligned}$$

This means that, once one discrete logarithm has been found, determining the next one is easier since we have already computed the values of the x_i.

The best of the methods to find the relations between the factorbase elements is the Number Field Sieve. This gives an overall running time of $O(L_p(1/3, c))$ for some constant c. This is roughly the same complexity as the algorithms to factor large numbers, although the real practical problem is that the matrix algorithms now need to work modulo $p-1$ and not modulo 2 as they did in the factoring algorithms.

The upshot of these sub-exponential methods is that the size of p for finite field discrete-logarithm-based systems needs to be of the same order of magnitude as a factoring modulus, i.e. $p \geq 2^{2048}$. Even though p has to be very large we still need to guard against generic attacks, hence $p-1$ should have a prime factor q of order greater than 2^{256}. In fact, for finite-field-based systems we usually work in the subgroup of $\mathbb{F}_p^*$ of order q.

In 2013, Antoine Joux and others showed that discrete logarithms in finite fields of characteristic two can be determined in quasi-polynomial time. This result is striking, but the methods used do not seem to generalize to higher-characteristic fields. This confirms the long-standing belief that discrete-logarithm-based cryptographic systems in fields of low characteristic should be avoided.

Chapter Summary

- We covered the DLP, DHP and DDH problems and the relationships between them.
- Due to the Pohlig–Hellman algorithm a hard discrete logarithm problem should be set in a group where the order has a large prime factor.
- Generic algorithms such as the Baby-Step/Giant-Step algorithm mean that to achieve the same security as a 128-bit block cipher, the size of the large prime factor of the group order should be at least 256 bits.
- The Baby-Step/Giant-Step algorithm is a generic algorithm and its running time can be absolutely bounded by $O(\sqrt{q})$, where q is the size of the large prime factor of $\#G$. However, the storage requirements of the Baby-Step/Giant-Step algorithm are also $O(\sqrt{q})$.
- There are a number of techniques, due to Pollard, based on deterministic random walks in a group. These are generic algorithms which require little storage but which solve the discrete logarithm problem in expected time $O(\sqrt{q})$.
- For finite fields a number of index calculus algorithms exist which run in sub-exponential time. These mean that one needs to take large finite fields $\mathbb{F}_{p^t}$ with $p^t \geq 2^{2048}$ to obtain a hard discrete logarithm problem. Due to the new quasi-polynomial-time attacks on low-characteristic fields we should select the characteristic to not be "too small".

Further Reading

There are a number of good surveys on the discrete logarithm problem. I would recommend the ones by McCurley and Odlyzko. For a more modern perspective see the article by Odlyzko, Pierrot and Joux.

K. McCurley. *The discrete logarithm problem.* In *Cryptology and Computational Number Theory*, Proc. Symposia in Applied Maths, Volume 42, 47–94, AMS, 1990.

A. Odlyzko. *Discrete logarithms: The past and the future.* Designs, Codes and Cryptography, **19**, 129–145, 2000.

A. Odlyzko, C. Pierrot and A. Joux. *The past, evolving present, and future of the discrete logarithm.* In *Open Problems in Mathematics and Computational Science*, Springer, 2014.

CHAPTER 4

Elliptic Curves

Chapter Goals

- To describe what an elliptic curve is.
- To explain the basic mathematics behind elliptic curve cryptography.
- To show how projective coordinates can be used to improve computational efficiency.
- To show how point compression can be used to improve communications efficiency.

4.1. Introduction

This chapter is devoted to introducing elliptic curves. Some of the more modern public key systems make use of elliptic curves since they can offer improved efficiency and bandwidth. Since much of this book can be read with just the understanding that an elliptic curve provides another finite abelian group in which one can pose a discrete logarithm problem, you may decide to skip this chapter on an initial reading.

Let K be any field. The projective plane $\mathbb{P}^2(K)$ over K is defined as the set of triples

$$(X, Y, Z)$$

where $X, Y, Z \in K$ are not all simultaneously zero. On these triples is defined an equivalence relation

$$(X, Y, Z) \equiv (X_1, Y_1, Z_1)$$

if there exists a $\lambda \in K$ such that

$$X = \lambda \cdot X_1,\ Y = \lambda \cdot Y_1 \text{ and } Z = \lambda \cdot Z_1.$$

So, for example, if $K = \mathbb{F}_7$, the finite field of seven elements, then the two points

$$(4, 1, 1) \text{ and } (5, 3, 3)$$

are equivalent. Such a triple is called a projective point.

An *elliptic curve* over K will be defined as the set of solutions in the projective plane $\mathbb{P}^2(K)$ of a homogeneous Weierstrass equation of the form

$$E:\ Y^2 \cdot Z + a_1 \cdot X \cdot Y \cdot Z + a_3 \cdot Y \cdot Z^2 = X^3 + a_2 \cdot X^2 \cdot Z + a_4 \cdot X \cdot Z^2 + a_6 \cdot Z^3,$$

with $a_1, a_2, a_3, a_4, a_6 \in K$. This equation is also referred to as the long Weierstrass form. Such a curve should be non-singular in the sense that, if the equation is written in the form $F(X, Y, Z) = 0$, then the partial derivatives of the curve equation

$$\partial F/\partial X,\ \partial F/\partial Y \text{ and } \partial F/\partial Z$$

should not vanish simultaneously at any point on the curve, i.e. the three simultaneous equations have no zero defined over the algebraic closure $\overline{K}$.

The set of K-rational points on E, i.e. the solutions in $\mathbb{P}^2(K)$ to the above equation, is denoted by $E(K)$. Notice that the curve has exactly one rational point with coordinate Z equal to zero, namely $(0, 1, 0)$. This is called the point at infinity, which will be denoted by $\mathcal{O}$.

4.1.1. The Affine Form: For convenience, we will most often use the affine version of the Weierstrass equation, given by

$$E:\ Y^2 + a_1 \cdot X \cdot Y + a_3 \cdot Y = X^3 + a_2 \cdot X^2 + a_4 \cdot X + a_6, \tag{5}$$

where $a_i \in K$; this is obtained by setting $Z = 1$ in the above equation. The K-rational points in the affine case are the solutions to E in K^2, plus the point at infinity $\mathcal{O}$. Although most protocols for elliptic-curve-based cryptography make use of the affine form of a curve, it is often computationally important to be able to switch to projective coordinates. Luckily this switch is easy:

- The point at infinity always maps to the point at infinity in either direction.
- To map a projective point (X, Y, Z) which is not at infinity, so $Z \neq 0$, to an affine point we simply compute $(X/Z, Y/Z)$.
- To map an affine point (X, Y), which is not at infinity, to a projective point we take a random non-zero $Z \in K$ and compute $(X \cdot Z, Y \cdot Z, Z)$.

As we shall see later it is often more convenient to use a slightly modified form of projective point where the projective point (X, Y, Z) represents the affine point $(X/Z^2, Y/Z^3)$, which equates to using the projective equation

$$E:\ Y^2 + a_1 \cdot X \cdot Y \cdot Z + a_3 \cdot Y \cdot Z^3 = X^3 + a_2 \cdot X^2 \cdot Z^2 + a_4 \cdot X \cdot Z^4 + a_6 \cdot Z^6.$$

4.1.2. Isomorphisms of Elliptic Curves: Given an elliptic curve defined by equation (5), it is useful to define the following constants for use in later formulae:

$$\begin{aligned}
b_2 &= a_1^2 + 4 \cdot a_2, \\
b_4 &= a_1 \cdot a_3 + 2 \cdot a_4, \\
b_6 &= a_3^2 + 4 \cdot a_6, \\
b_8 &= a_1^2 \cdot a_6 + 4 \cdot a_2 \cdot a_6 - a_1 \cdot a_3 \cdot a_4 + a_2 \cdot a_3^2 - a_4^2, \\
c_4 &= b_2^2 - 24 \cdot b_4, \\
c_6 &= -b_2^3 + 36 \cdot b_2 \cdot b_4 - 216 \cdot b_6.
\end{aligned}$$

The discriminant of the curve is defined as

$$\Delta = -b_2^2 \cdot b_8 - 8 \cdot b_4^3 - 27 \cdot b_6^2 + 9 \cdot b_2 \cdot b_4 \cdot b_6.$$

When the characteristic of the field $\mathsf{char}K \neq 2, 3$ the discriminant can also be expressed as

$$\Delta = (c_4^3 - c_6^2)/1728.$$

Notice that $1728 = 2^6 \cdot 3^3$ so, if the characteristic of the underlying finite field is not equal to 2 or 3, dividing by this latter quantity makes sense. A curve is then non-singular if and only if $\Delta \neq 0$; from now on we shall assume that $\Delta \neq 0$ in all our discussions. When $\Delta \neq 0$, the j-invariant of the curve is defined as

$$j(E) = c_4^3/\Delta.$$

As an example, which we shall use throughout this chapter, we consider the elliptic curve

$$E:\ Y^2 = X^3 + X + 3$$

defined over the field $\mathbb{F}_7$. Computing the various quantities above we find that we have

$$\Delta = 3 \text{ and } j(E) = 5.$$

The j-invariant is closely related to the notion of elliptic curve isomorphism. Two elliptic curves defined by Weierstrass equations E (with variables X, Y) and E_1 (with variables X_1, Y_1) are isomorphic over K if and only if there exist constants $r, s, t \in K$ and $u \in K^*$, such that the change of variables

$$X = u^2 \cdot X_1 + r\ , \quad Y = u^3 \cdot Y_1 + s \cdot u^2 \cdot X_1 + t$$

transforms E into E_1. This transformation might look special, but arises from the requirements that the isomorphism should map the two points at infinity to themselves, and should keep unchanged the structure of the affine equation (5).

Such an isomorphism defines a bijection between the set of rational points in E and the set of rational points in E_1. Notice that isomorphism is defined relative to the field K. As an example consider again the elliptic curve

$$E:\ Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. Now make the change of variables defined by $[u, r, s, t] = [2, 3, 4, 5]$, i.e.

$$X = 4 \cdot X_1 + 3 \text{ and } Y = Y_1 + 2 \cdot X_1 + 5.$$

We then obtain the isomorphic curve

$$E_1:\ Y_1^2 + 4 \cdot X_1 \cdot Y_1 + 3 \cdot Y_1 = X_1^3 + X_1 + 1,$$

and we have

$$j(E) = j(E_1) = 5.$$

Curve isomorphism is an equivalence relation. The following lemma establishes the fact that, over the algebraic closure $\overline{K}$, the j-invariant characterizes the equivalence classes in this relation.

Lemma 4.1. *Two elliptic curves that are isomorphic over K have the same j-invariant. Conversely, two curves with the same j-invariant are isomorphic over the algebraic closure $\overline{K}$.*

But curves with the same j-invariant may not necessarily be isomorphic over the ground field. For example, consider the elliptic curve, also over $\mathbb{F}_7$,

$$E_2:\ Y_2^2 = X_2^3 + 4 \cdot X_2 + 4.$$

This has j-invariant equal to 5 so it is isomorphic to E over $\overline{\mathbb{F}_7}$, but it is not isomorphic over $\mathbb{F}_7$ since the change of variable required is given by

$$X = 3 \cdot X_2 \text{ and } Y = \sqrt{6} \cdot Y_2.$$

However, $\sqrt{6} \notin \mathbb{F}_7$. Hence, we say both E and E_2 are defined over $\mathbb{F}_7$, but they are isomorphic over $\mathbb{F}_{7^2} = \mathbb{F}_7[\sqrt{6}] \subset \overline{\mathbb{F}_7}$.

4.2. The Group Law

Assume, for the moment, that $\mathsf{char}K \neq 2, 3$, and consider the change of variables given by

$$X = X_1 - \frac{b_2}{12},$$
$$Y = Y_1 - \frac{a_1}{2} \cdot \left(X_1 - \frac{b_2}{12} \right) - \frac{a_3}{2}.$$

This change of variables transforms the long Weierstrass form given in equation (5) to the equation of an isomorphic curve given in short Weierstrass form,

$$E:\ Y^2 = X^3 + a \cdot X + b,$$

for some $a, b \in K$. One can then define a group law on an elliptic curve using the chord-tangent process.

The chord process is defined as follows; see Figure 4.1 for a diagrammatic description. Let P and Q be two distinct points on E. The straight line joining P and Q must intersect the curve at one further point, say R, since we are intersecting a line with a cubic curve. The point R will also be defined over the same field of definition as the curve and the two points P and Q. If we then reflect R in the x-axis we obtain another point over the same field which we shall call $P + Q$.

The tangent process is given diagrammatically in Figure 4.2 or as follows, for a point P on the curve E. We take the tangent to the curve at P; such a line must intersect E in at most one other

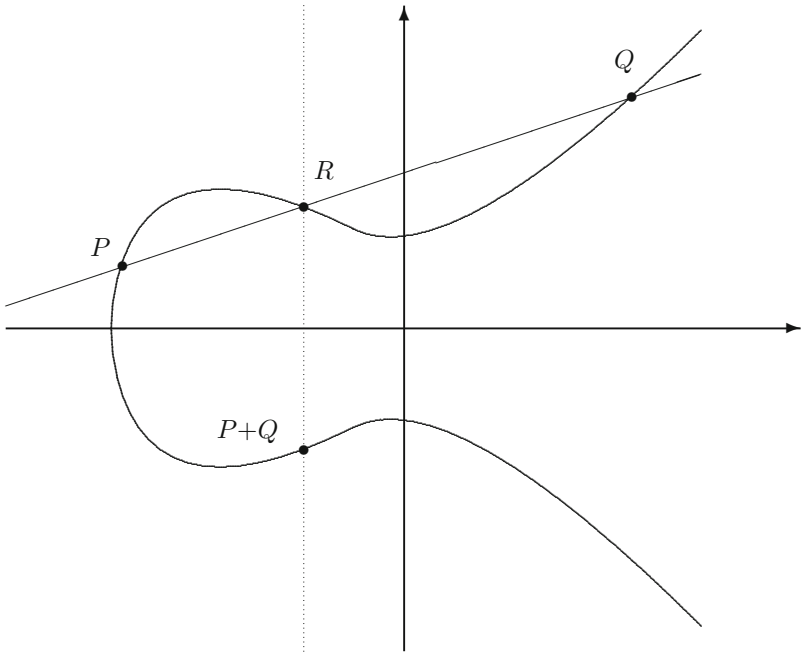


FIGURE 4.1. Adding two points on an elliptic curve

point, say R, as the elliptic curve E is defined by a cubic equation. Again we reflect R in the x-axis to obtain a point which we call $[2]P = P + P$. If the tangent to the point is vertical, it "intersects" the curve at the point at infinity and $P + P = \mathcal{O}$, and P is said to be a point of order 2.

One can show that the chord-tangent process turns E into an abelian group with the point at infinity $\mathcal{O}$ being the identity. The above definition can easily be extended to the long Weierstrass form (and so to characteristic two and three). One simply changes the definition by replacing "reflection in the x-axis" by "reflection in the line $Y = a_1 \cdot X + a_3$". In addition a little calculus will result in explicit algebraic formulae for the chord-tangent process. This is necessary since drawing diagrams as above is not really allowed in a field of finite characteristic. The algebraic formulae are summarized in the following lemma.

Lemma 4.2. *Let E denote an elliptic curve given by*

$$E: \; Y^2 + a_1 \cdot X \cdot Y + a_3 \cdot Y = X^3 + a_2 \cdot X^2 + a_4 \cdot X + a_6$$

and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ denote points on the curve. Then

$$-P_1 = (x_1, -y_1 - a_1 \cdot x_1 - a_3).$$

Set

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$
$$\mu = \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1}$$

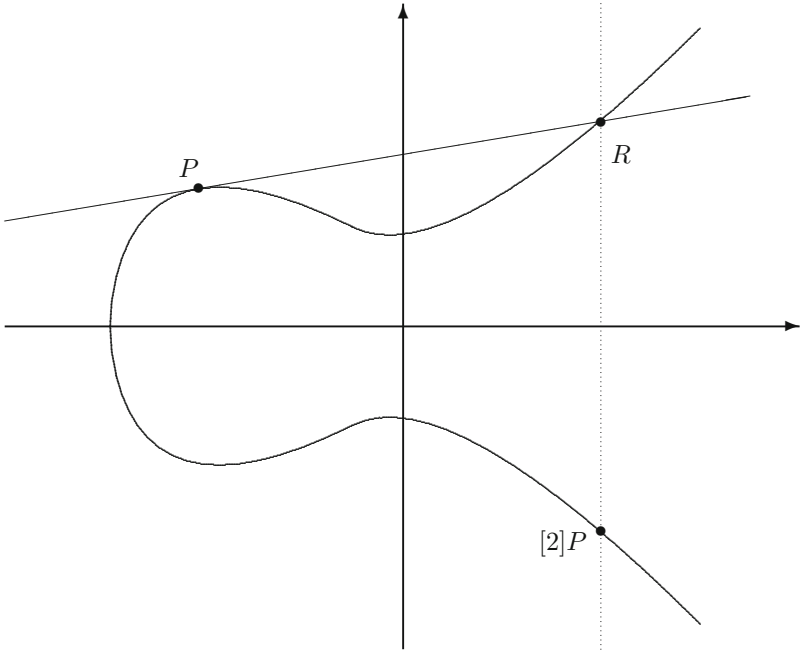


FIGURE 4.2. Doubling a point on an elliptic curve

when $x_1 \neq x_2$*, and set*

$$\lambda = \frac{3 \cdot x_1^2 + 2 \cdot a_2 \cdot x_1 + a_4 - a_1 \cdot y_1}{2 \cdot y_1 + a_1 \cdot x_1 + a_3},$$
$$\mu = \frac{-x_1^3 + a_4 \cdot x_1 + 2 \cdot a_6 - a_3 \cdot y_1}{2 \cdot y_1 + a_1 \cdot x_1 + a_3}$$

when $x_1 = x_2$ *and* $P_2 \neq -P_1$*. If*

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O}$$

then x_3 *and* y_3 *are given by the formulae*

$$x_3 = \lambda^2 + a_1 \cdot \lambda - a_2 - x_1 - x_2,$$
$$y_3 = -(\lambda + a_1) \cdot x_3 - \mu - a_3.$$

The elliptic curve isomorphisms described earlier then become group isomorphisms as they respect the group structure.

4.2.1. The Elliptic Curve Discrete Logarithm Problem (ECDLP): For a positive integer m we let $[m]$ denote the multiplication-by-m map from the curve to itself. This map takes a point P to

$$P + P + \cdots + P,$$

where we have m summands. This map is the basis of elliptic curve cryptography, since whilst it is easy to compute, it is believed to be hard to invert, i.e. given $P = (x, y)$ and $[m]P = (x', y')$ it is

hard to compute m. Of course this statement of hardness assumes a well-chosen elliptic curve etc., something we will return to later.

Example: We end this section with an example of the elliptic curve group law. Again we take our elliptic curve

$$E:\ Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. It turns out there are six points on this curve given by

$$\mathcal{O},\ (4,1),\ (6,6),\ (5,0),\ (6,1) \text{ and } (4,6).$$

These form a group with the group law being given by the following table, which is computed using the addition formulae given above.

$+$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$
$\mathcal{O}$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$
$(4,1)$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$
$(6,6)$	$(6,6)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$
$(5,0)$	$(5,0)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$
$(6,1)$	$(6,1)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$
$(4,6)$	$(4,6)$	$\mathcal{O}$	$(4,1)$	$(6,6)$	$(5,0)$	$(6,1)$

As an example of the multiplication-by-m map, if we let $P = (4,1)$ then we have

$$[2]P = (6,6),\ [3]P = (5,0),\ [4]P = (6,1),\ [5]P = (4,6),\ [6]P = \mathcal{O}.$$

So we see in this example that $E(\mathbb{F}_7)$ is a finite cyclic abelian group of order six generated by the point P. For all elliptic curves over finite fields the group is always finite and it is also highly likely to be cyclic (or "nearly" cyclic).

4.3. Elliptic Curves over Finite Fields

Over a finite field $\mathbb{F}_q$, the number of rational points on a curve is finite, and its size will be denoted by $\#E(\mathbb{F}_q)$. The expected number of points on the curve is around $q+1$ and if we set

$$\#E(\mathbb{F}_q) = q + 1 - t$$

then the value t is called the *trace of Frobenius* at q. A first approximation to the order of $E(\mathbb{F}_q)$ is given by the following well-known theorem of Hasse.

Theorem 4.3 (H. Hasse, 1933)**.** *The trace of Frobenius satisfies*

$$|t| \leq 2 \cdot \sqrt{q}.$$

Consider our example of

$$E:\ Y^2 = X^3 + X + 3$$

then recall this has six points over the field $\mathbb{F}_7$, and so the associated trace of Frobenius is equal to 2, which is less than $2 \cdot \sqrt{q} = 2 \cdot \sqrt{7} = 5.29$.

The qth-power Frobenius map, on an elliptic curve E defined over $\mathbb{F}_q$, is given by

$$\varphi : \begin{cases} E(\overline{\mathbb{F}}_q) \longrightarrow E(\overline{\mathbb{F}}_q) \\ (x,y) \longmapsto (x^q, y^q) \\ \mathcal{O} \longmapsto \mathcal{O}. \end{cases}$$

The map φ sends points on E to points on E, no matter what the field of definition of the point is. In addition the map φ respects the group law in that

$$\varphi(P+Q) = \varphi(P) + \varphi(Q).$$

In other words the map φ is a group endomorphism of E over the algebraic closure of $\mathbb{F}_q$, which is denoted by $\overline{\mathbb{F}}_q$, referred to as the Frobenius endomorphism. The trace of Frobenius t and the Frobenius endomorphism φ are linked by the equation

$$\varphi^2 - [t]\varphi + [q] = [0].$$

Hence, for any point $P = (x, y)$ on the curve, we have

$$(x^{q^2}, y^{q^2}) - [t](x^q, y^q) + [q](x, y) = \mathcal{O},$$

where addition and subtraction denote curve operations.

As was apparent from the earlier discussion, the cases char $K = 2, 3$ often require separate treatment. Practical implementations of elliptic curve cryptosystems are usually based on either $\mathbb{F}_{2^n}$, i.e. characteristic two, or $\mathbb{F}_p$ for large primes p. Therefore, in the remainder of this chapter we will focus on fields of characteristic two and $p > 3$, and will omit the separate treatment of the case $\mathsf{char}K = 3$. Most arguments, though, carry across easily to characteristic three, with modifications that are well documented in the literature.

4.3.1. Curves over Fields of Characteristic $p > 3$: Assume that our finite field is given by $K = \mathbb{F}_q$, where $q = p^n$ for a prime $p > 3$ and an integer $n \geq 1$. As mentioned, the curve equation in this case can be simplified to the short Weierstrass form

$$E: \ Y^2 = X^3 + a \cdot X + b.$$

The discriminant of the curve then reduces to $\Delta = -16 \cdot (4 \cdot a^3 + 27 \cdot b^2)$, and its j-invariant to $j(E) = -1728 \cdot (4 \cdot a)^3/\Delta$. The formulae for the group law in Lemma 4.2 also simplify to

$$-P_1 = (x_1, -y_1),$$

and if

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O},$$

then x_3 and y_3 are given by the formulae

$$x_3 = \lambda^2 - x_1 - x_2,$$
$$y_3 = (x_1 - x_3) \cdot \lambda - y_1,$$

where if $x_1 \neq x_2$ we set

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$

and if $x_1 = x_2$, $y_1 \neq 0$ we set

$$\lambda = \frac{3 \cdot x_1^2 + a}{2 \cdot y_1}.$$

4.3.2. Curves over Fields of Characteristic Two: We now specialize to the case of finite fields where $q = 2^n$ with $n \geq 1$. In this case, the expression for the j-invariant reduces to $j(E) = a_1^{12}/\Delta$. In characteristic two, the condition $j(E) = 0$, i.e. $a_1 = 0$, is equivalent to the curve being supersingular. As mentioned earlier, this very special type of curve is avoided in cryptography. We assume, therefore, that $j(E) \neq 0$.

Under these assumptions, a representative for each isomorphism class of elliptic curves over $\mathbb{F}_q$ is given by

$$E: \ Y^2 + X \cdot Y = X^3 + a_2 \cdot X^2 + a_6, \tag{6}$$

where $a_6 \in \mathbb{F}_q^*$ and $a_2 \in \{0, \gamma\}$ with γ a fixed element in $\mathbb{F}_q$ such that $\mathrm{Tr}_{q|2}(\gamma) = 1$, where $\mathrm{Tr}_{q|2}$ is the absolute trace

$$\mathrm{Tr}_{2^n|2}(\alpha) = \sum_{i=0}^{n-1} \alpha^{2^i}.$$

The formulae for the group law in Lemma 4.2 then simplify to

$$-P_1 = (x_1, y_1 + x_1),$$

and if

$$P_3 = (x_3, y_3) = P_1 + P_2 \neq \mathcal{O},$$

then x_3 and y_3 are given by the formulae

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a_2 + x_1 + x_2, \\ y_3 &= (\lambda + 1) \cdot x_3 + \mu \\ &= (x_1 + x_3) \cdot \lambda + x_3 + y_1, \end{aligned}$$

where if $x_1 \neq x_2$ we set

$$\begin{aligned} \lambda &= \frac{y_2 + y_1}{x_2 + x_1}, \\ \mu &= \frac{y_1 \cdot x_2 + y_2 \cdot x_1}{x_2 + x_1} \end{aligned}$$

and if $x_1 = x_2 \neq 0$ we set

$$\begin{aligned} \lambda &= \frac{x_1^2 + y_1}{x_1}, \\ \mu &= x_1^2. \end{aligned}$$

4.4. Projective Coordinates

One of the problems with the above formulae for the group laws, given in both large and even characteristic, is that at some stage they involve a division operation. Division in finite fields is considered to be an expensive operation, since it usually involves some variant of the extended Euclidean algorithm, which although of approximately the same complexity as multiplication can usually not be implemented as efficiently.

To avoid these division operations one can use projective coordinates. Here one writes the elliptic curve using three variables (X, Y, Z) instead of just (X, Y). Instead of using the projective representation given at the start of this chapter we instead use one where the curve is written as

$$E: \ Y^2 + a_1 \cdot X \cdot Y \cdot Z + a_2 \cdot Y \cdot Z^4 = X^3 + a_2 \cdot X^2 \cdot Z^2 + a_4 \cdot X \cdot Z^4 + a_6 \cdot Z^6.$$

The point at infinity is still denoted by $(0, 1, 0)$, but now the map from projective to affine coordinates is given by

$$(X, Y, Z) \longmapsto (X/Z^2, Y/Z^3).$$

This choice of projective coordinates is made to provide a more efficient arithmetic operation.

4.4.1. Large Prime Characteristic: The formulae for point addition when our elliptic curve is written as

$$E: \ Y^2 = X^3 + a \cdot X \cdot Z^4 + b \cdot Z^6$$

are now given by the law

$$(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$$

where (X_3, Y_3, Z_3) are derived from the formulae

$$\begin{aligned}
\lambda_1 &= X_1 \cdot Z_2^2, & \lambda_2 &= X_2 \cdot Z_1^2,\\
\lambda_3 &= \lambda_1 - \lambda_2, & \lambda_4 &= Y_1 \cdot Z_2^3,\\
\lambda_5 &= Y_2 \cdot Z_1^3, & \lambda_6 &= \lambda_4 - \lambda_5,\\
\lambda_7 &= \lambda_1 + \lambda_2, & \lambda_8 &= \lambda_4 + \lambda_5,\\
Z_3 &= Z_1 \cdot Z_2 \cdot \lambda_3, & X_3 &= \lambda_6^2 - \lambda_7 \cdot \lambda_3^2,\\
\lambda_9 &= \lambda_7 \cdot \lambda_3^2 - 2 \cdot X_3, & Y_3 &= (\lambda_9 \cdot \lambda_6 - \lambda_8 \cdot \lambda_3^3)/2.
\end{aligned}$$

Notice the avoidance of any division operation, bar division by 2 which can be easily accomplished by multiplication of the precomputed value of $2^{-1} \pmod p$. Doubling a point,

$$(X_3, Y_3, Z_3) = [2](X_1, Y_1, Z_1),$$

can be accomplished using the formulae

$$\begin{aligned}
\lambda_1 &= 3 \cdot X_1^2 + a \cdot Z_1^4, & Z_3 &= 2 \cdot Y_1 \cdot Z_1,\\
\lambda_2 &= 4 \cdot X_1 \cdot Y_1^2, & X_3 &= \lambda_1^2 - 2 \cdot \lambda_2,\\
\lambda_3 &= 8 \cdot Y_1^4, & Y_3 &= \lambda_1 \cdot (\lambda_2 - X_3) - \lambda_3.
\end{aligned}$$

4.4.2. Even Characteristic: In even characteristic we write our elliptic curve in the form

$$E:\ Y^2 + X \cdot Y \cdot Z = X^3 + a_2 \cdot X^2 \cdot Z^2 + a_6 \cdot Z^6.$$

Point addition,

$$(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$$

is now accomplished using the recipe

$$\begin{aligned}
\lambda_1 &= X_1 \cdot Z_2^2, & \lambda_2 &= X_2 \cdot Z_1^2,\\
\lambda_3 &= \lambda_1 + \lambda_2, & \lambda_4 &= Y_1 \cdot Z_2^3,\\
\lambda_5 &= Y_2 \cdot Z_1^3, & \lambda_6 &= \lambda_4 + \lambda_5,\\
\lambda_7 &= Z_1 \cdot \lambda_3, & \lambda_8 &= \lambda_6 \cdot X_2 + \lambda_7 \cdot Y_2,\\
Z_3 &= \lambda_7 \cdot Z_2, & \lambda_9 &= \lambda_6 + Z_3,\\
X_3 &= a_2 \cdot Z_3^2 + \lambda_6 \cdot \lambda_9 + \lambda_3^3, & Y_3 &= \lambda_9 \cdot X_3 + \lambda_8 \cdot \lambda_7^2.
\end{aligned}$$

Doubling is performed using

$$\begin{aligned}
Z_3 &= X_1 \cdot Z_1^2, & X_3 &= (X_1 + d_6 \cdot Z_1^2)^4,\\
\lambda &= Z_3 + X_1^2 + Y_1 \cdot Z_1, & Y_3 &= X_1^4 \cdot Z_3 + \lambda \cdot X_3,
\end{aligned}$$

where $d_6 = \sqrt[4]{a_6}$. Notice how in both even and odd characteristic we have avoided a division operation when performing curve operations.

4.5. Point Compression

In many cryptographic protocols we need to store or transmit an elliptic curve point. Using affine coordinates this can be accomplished using two field elements, i.e. by transmitting x and then y. However, one can do better using a technique called point compression. Point compression is based on the observation that for every x-coordinate on the curve there are at most two corresponding y-coordinates. Hence, we can represent a point by storing the x-coordinate along with a bit b to say which value of the y-coordinate we should take. All that remains to decide is how to compute the bit b and how to reconstruct the y-coordinate given the x-coordinate and the bit b.

Large Prime Characteristic: For elliptic curves over fields of large prime characteristic we notice that if $\alpha \in \mathbb{F}_p^*$ is a square, then the two square roots $\pm\beta$ of α have different parities when represented as integers in the range $[1, \ldots, p-1]$. This is because $-\beta = p - \beta$. Hence, as the bit b we choose the parity of the y-coordinate. Given (x, b), we can reconstruct y by computing

$$\beta = \sqrt{x^3 + a \cdot x + b} \pmod{p}.$$

If the parity of β is equal to b we set $y = \beta$, otherwise we set $y = p - \beta$. If $\beta = 0$ then no matter which value of b we have we set $y = 0$.

As an example consider the curve

$$E: \ Y^2 = X^3 + X + 3$$

over the field $\mathbb{F}_7$. Then the points $(4, 1)$ and $(4, 6)$ which in bits we need to represent as

$$(0b100, 0b001) \text{ and } (0b100, 0b110),$$

i.e. requiring six bits for each point, can be represented as

$$(0b100, 0b1) \text{ and } (0b100, 0b0),$$

where we only use four bits for each point. In larger, cryptographically interesting, examples the advantage becomes more pronounced. For example consider the curve with the same coefficients but over the finite field $\mathbb{F}_p$ where

$$p = 1\,125\,899\,906\,842\,679 = 2^{50} + 55$$

then the point

$$(1\,125\,899\,906\,842\,675, \ \ 245\,132\,605\,757\,739)$$

can be represented by the integers

$$(1\,125\,899\,906\,842\,675, \ \ 1).$$

So instead of requiring 102 bits we only require 52 bits.

Even Characteristic: In even characteristic we need to be slightly more clever. Suppose we are given a point $P = (x, y)$ on the elliptic curve

$$E: \ Y^2 + X \cdot Y = X^3 + a_2 \cdot X + a_6.$$

If $y = 0$ then we set $b = 0$, otherwise we compute

$$z = y/x$$

and let b denote the least significant bit of z. To recover y given (x, b), for $x \neq 0$, we set

$$\alpha = x + a_2 + \frac{a_6}{x^2}$$

and let β denote a solution of

$$z^2 + z = \alpha.$$

Then if the least significant bit of β is equal to b we set $y = x \cdot \beta$, otherwise we set $y = x \cdot (\beta + 1)$. To see why this works notice that if (x, y) is a solution of

$$E: \ Y^2 + XY = X^3 + a_2 \cdot X^2 + a_6$$

then $(x, y/x)$ and $(x, 1 + y/x)$ are the two solutions of

$$Z^2 + Z = X + a_2 + \frac{a_6}{X^2}.$$

4.6. Choosing an Elliptic Curve

One of the advantages of elliptic curves is that there is a very large number of possible groups. One can choose both the finite field and the coefficients of the curve. In addition finding elliptic curves with the correct cryptographic properties to make the systems using them secure is relatively easy; we just have to know which curves to avoid.

For any elliptic curve and any finite field the group order $\#E(\mathbb{F}_q)$ can be computed in polynomial time. But this is usually done via a complicated algorithm that we cannot go into in this book. Hence, you should just remember that computing the group order is computationally easy. As we saw in Chapter 3, when considering algorithms to solve discrete logarithm problems, knowing the group order is important in understanding how secure a group is. For some elliptic curves computing the group order is easy; in particular supersingular curves. The curve $E(\mathbb{F}_q)$ is said to be supersingular if the characteristic p divides the trace of Frobenius, t. If $q = p$ then this means that $E(\mathbb{F}_p)$ has $p + 1$ points since we must have $t = 0$. For other finite fields the possible values of t corresponding to supersingular elliptic curves are given by, where $q = p^f$,

- f odd: $t = 0$, $t^2 = 2q$ and $t^2 = 3q$.
- f even: $t^2 = 4q$, $t^2 = q$ if $p = 1 \pmod 3$ and $t = 0$ if $p \neq 1 \pmod 4$.

For elliptic curves there are no known sub-exponential methods for the discrete logarithm problem, except in certain special cases. There are three particular classes of curves which, under certain conditions, will prove to be cryptographically weak:

- The curve $E(\mathbb{F}_q)$ is said to be anomalous if its trace of Frobenius is one, giving $\#E(\mathbb{F}_q) = q$. These curves are weak when $q = p$, the field characteristic. In this case there is an algorithm to solve the discrete logarithm problem which requires $O(\log p)$ elliptic curve operations.
- For any q we must choose curves for which there is no small number t such that r divides $q^t - 1$, where r is the large prime factor of $\#E(\mathbb{F}_q)$. This alas eliminates the supersingular curves and a few others. In this case there is a simple computable mapping from the elliptic curve discrete logarithm problem to the discrete logarithm problem in the finite field $\mathbb{F}_{q^t}$. Hence, in this case we obtain a sub-exponential method for solving the elliptic curve discrete logarithm problem.
- If $q = 2^n$ then we usually assume that n is prime to avoid the possibility of certain attacks based on the concept of "Weil descent".

One should treat these three special cases much like one treats the generation of large integers for the RSA algorithm. Due to the $P - 1$ factoring method one often makes RSA moduli $N = p \cdot q$ such that p is a so-called safe prime of the form $2p_1 + 1$. Another special RSA-based case is that we almost always use RSA with a modulus having two prime factors, rather than three or four. This is because moduli with two prime factors appear to be the hardest to factor.

It turns out that the only known practical method to solve the discrete logarithm problem in general elliptic curves is the parallel version of Pollard's Rho method given in Chapter 3. Thus we need to choose a curve such that the group order $\#E(\mathbb{F}_q)$ is divisible by a large prime number r, and for which the curve is not considered weak by the above considerations. Hence, from now on we suppose the elliptic curve E is defined over the finite field $\mathbb{F}_q$ and we have

$$\#E(\mathbb{F}_q) = h \cdot r$$

where r is a "large" prime number and h is a small number called the cofactor. By Hasse's Theorem 4.3 the value of $\#E(\mathbb{F}_q)$ is close to q so we typically choose a curve with r close to q, i.e. we choose a curve E so that $h = 1, 2$ or 4.

Since the best general algorithm known for the elliptic curve discrete logarithm problem is the parallel Pollard's Rho method, which has complexity $O(\sqrt{r})$, which is about $O(\sqrt{q})$, to achieve the same security as a 128-bit block cipher we need to take $q \approx 2^{256}$, which is a lot smaller than the

field size recommended for systems based on the discrete logarithm problems in a finite field. This results in the reduced bandwidth and computational times of elliptic curve systems.

Chapter Summary

- Elliptic curves over finite fields are another example of a finite abelian group. There are many such groups since we are free to choose both the curve and the field.
- For cryptography we need to be able to compute the number of elements in the group. Although this is done using a complicated algorithm, it can be done in polynomial time.
- One should usually avoid supersingular and anomalous curves in cryptographic applications.
- Efficient algorithms for the group law can be produced by using projective coordinates. These algorithms avoid the need for costly division operations in the underlying finite field.
- To save bandwidth and space it is possible to efficiently compress elliptic curve points (x, y) down to x and a single bit b. The uncompression can also be performed efficiently.
- For elliptic curves there are no sub-exponential algorithms known for the discrete logarithm problem, except in very special cases. Hence, the only practical general algorithm to solve the discrete logarithm problem on an elliptic curve is the parallel Pollard's Rho method.

Further Reading

Those who wish to learn more about elliptic curves in general may try the textbook by Silverman (which is really aimed at mathematics graduate students). Those who are simply interested in the cryptographic applications of elliptic curves and the associated algorithms and techniques may see the book by Blake, Seroussi and Smart and its follow-up book.

I.F. Blake, G. Seroussi and N.P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

I.F. Blake, G. Seroussi and N.P. Smart. *Advances in Elliptic Curve Cryptography.* Cambridge University Press, 2004.

J.H. Silverman. *The Arithmetic of Elliptic Curves.* Springer, 1985.

CHAPTER 5

Lattices

Chapter Goals

- To describe lattice basis reduction algorithms and give some examples of how they are used to break cryptographic systems.
- To introduce the hard problems of SVP, CVP, BDD and their various variations.
- To introduce q-ary lattices.
- To explain the technique of Coppersmith for finding small roots of modular polynomial equations.

5.1. Lattices and Lattice Reduction

In this chapter we present the concept of a lattice. Traditionally in cryptography lattices have been used in cryptanalysis to break systems, and we shall see applications of this in Chapter 15. However, recently they have also been used to construct cryptographic systems with special properties, as we shall see in Chapter 17.

5.1.1. Vector Spaces: Before presenting lattices, and the technique of lattice basis reduction, we first need to recap some basic linear algebra. Suppose $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is an n-dimensional real vector, i.e. for all i we have $x_i \in \mathbb{R}$. The set of all such vectors is denoted $\mathbb{R}^n$. On two such vectors we can define an *inner product*

$$\langle \mathbf{x}, \mathbf{y} \rangle = x_1 \cdot y_1 + x_2 \cdot y_2 + \cdots + x_n \cdot y_n,$$

which is a function from pairs of n-dimensional vectors to the real numbers. You probably learnt at school that two vectors $\mathbf{x}$ and $\mathbf{y}$ are orthogonal, or meet at right angles, if and only if we have

$$\langle \mathbf{x}, \mathbf{y} \rangle = 0.$$

Given the inner product we can then define the size, or length, of a vector by

$$\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

This length corresponds to the intuitive notion of length of vectors; in particular the length satisfies a number of properties.

- $\|\mathbf{x}\| \geq 0$, with equality if and only if $\mathbf{x}$ is the zero vector.
- Triangle inequality: For two n-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|.$$

- Scaling: For a vector x and a real number a

$$\|a \cdot \mathbf{x}\| = |a| \cdot \|\mathbf{x}\|.$$

A set of vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ in $\mathbb{R}^n$ is called linearly independent if the equation

$$a_1 \cdot \mathbf{b}_1 + \cdots + a_m \cdot \mathbf{b}_m = 0,$$

for real numbers a_i, implies that all a_i are equal to zero. If the set is linearly independent then we must have $m \leq n$. Suppose we have a set of m linearly independent vectors, $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. We can look at the set of all real linear combinations of these vectors,

$$V = \left\{ \sum_{i=1}^{m} a_i \cdot \mathbf{b}_i : a_i \in \mathbb{R} \right\}.$$

This is a vector subspace of $\mathbb{R}^n$ of dimension m and the set $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ is called a basis of this subspace. If we form the matrix B with ith column of B being equal to $\mathbf{b}_i$ for all i then we have

$$V = \{B \cdot \mathbf{a} : \mathbf{a} \in \mathbb{R}^m\}.$$

The matrix B is called the basis matrix.

5.1.2. The Gram–Schmidt Process: Every subspace V has a large number of possible basis matrices. Given one such basis it is often required to produce a basis with certain prescribed nice properties. Often in applications throughout science and engineering one requires a basis which is pairwise orthogonal, i.e.

$$\langle \mathbf{b}_i, \mathbf{b}_j \rangle = 0$$

for all $i \neq j$. Luckily there is a well-known method which takes one basis, $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ and produces a basis $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_m^*\}$ which is pairwise orthogonal. This method is called the Gram–Schmidt process and the basis $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_m^*\}$ produced from $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ via this process is called the Gram–Schmidt basis corresponding to $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. One computes the $\mathbf{b}_i^*$ from the $\mathbf{b}_i$ via the recursive equations

$$\mu_{i,j} \leftarrow \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}, \text{ for } 1 \leq j < i \leq n,$$

$$\mathbf{b}_i^* \leftarrow \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \cdot \mathbf{b}_j^*.$$

For example if we have

$$\mathbf{b}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \text{ and } \mathbf{b}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

then we compute

$$\mathbf{b}_1^* \leftarrow \mathbf{b}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix},$$

$$\mathbf{b}_2^* \leftarrow \mathbf{b}_2 - \mu_{2,1} \cdot \mathbf{b}_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \frac{1}{2} \cdot \begin{pmatrix} 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

since

$$\mu_{2,1} = \frac{\langle \mathbf{b}_2, \mathbf{b}_1^* \rangle}{\langle \mathbf{b}_1^*, \mathbf{b}_1^* \rangle} = \frac{2}{4} = \frac{1}{2}.$$

Notice how we have $\langle b_1^*, b_2^* \rangle = 0$, so the new Gram–Schmidt basis is orthogonal.

5.1.3. Lattices: A *lattice* L is like the vector subspace V above, but given a set of basis vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ instead of taking the *real* linear combinations of the $\mathbf{b}_i$ we are only allowed to take the *integer* linear combinations of the $\mathbf{b}_i$,

$$L = \left\{ \sum_{i=1}^{m} a_i \cdot \mathbf{b}_i : a_i \in \mathbb{Z} \right\} = \{B \cdot \mathbf{a} : \mathbf{a} \in \mathbb{Z}^m\}.$$

The set $\{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ is still called the set of basis vectors and the matrix B is still called a basis matrix. To see why lattices are called lattices, consider the lattice L generated by the two vectors

$$\mathbf{b}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \text{ and } \mathbf{b}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

This is the set of all vectors of the form

$$\begin{pmatrix} 2 \cdot x + y \\ y \end{pmatrix},$$

where $x, y \in \mathbb{Z}$. If one plots these points in the plane then one sees that these points form a two-dimensional lattice. See for example Figure 5.1.

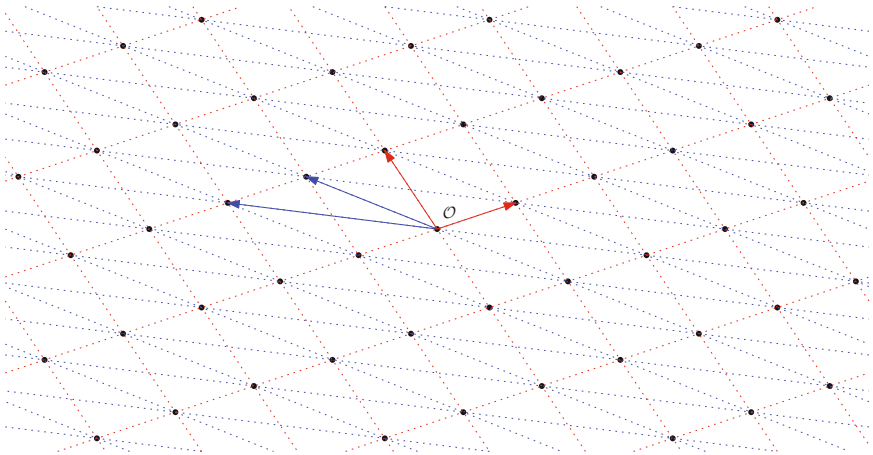


FIGURE 5.1. A lattice with two bases marked. A "nice" one in red, and a "bad" one in blue

A lattice is a discrete version of a vector subspace. Since it is discrete there is a well-defined smallest element, bar the trivially small element of the zero vector of course. This allows us to define the non-zero minimum of any lattice L, which we denote by

$$\lambda_1(L) := \min\{\|\mathbf{x}\| : \mathbf{x} \in L, \mathbf{x} \neq \mathbf{0}\}.$$

We can also define the successive minima $\lambda_i(L)$, which are defined as the smallest radius r such that the n-dimensional ball of radius r centred on the origin contains i linearly independent lattice points. Many tasks in computing, and especially cryptography, can be reduced to trying to determine the smallest non-zero vector in a lattice. We shall see some of these applications later, but before continuing with our discussion of lattices in general we pause to note that it is generally considered to be a hard problem to determine the smallest non-zero vector in an arbitrary lattice. Later we shall see that, whilst this problem is hard in general, it is in fact easy in low dimension, a situation which we shall use to our advantage later on.

Just as with vector subspaces, given a lattice basis one could ask, is there a nicer basis? For example in Figure 5.1 the red basis is certainly "more orthogonal" than the blue basis. Suppose B is a basis matrix for a lattice L. To obtain another basis matrix B' the only operation we are allowed to use is post-multiplication of B by a uni-modular integer matrix. This means we must have

$$B' = B \cdot U$$

for some integer matrix U with $\det(U) = \pm 1$. This means that the absolute value of the determinant of a basis matrix of a lattice is an invariant of the lattice, i.e. it does not depend on the choice of basis. Given a basis matrix B for a lattice L, we call

$$\Delta(L) = |\det(B^t \cdot B)|^{1/2}$$

the *discriminant* of the lattice. If L is a lattice of full rank, i.e. B is a square matrix, then we have

$$\Delta(L) = |\det(B)|.$$

The value $\Delta(L)$ represents the volume of the fundamental parallelepiped of the lattice bases; see for example Figure 5.2, which presents a lattice with two parallelepipeds marked. Each parallelepiped has the same volume, as both correspond to a basis of the lattice. If the lattice is obvious from the context we just write Δ. From now on we shall only consider full-rank lattices, and hence $n = m$, and our basis matrices will be square.

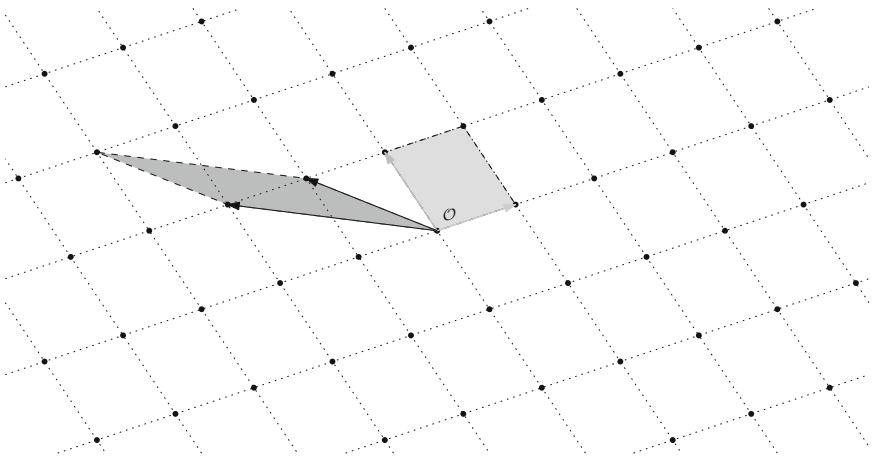


FIGURE 5.2. A lattice with two fundamental parallelepipeds marked

Hermite showed that there is an absolute constant γ_n, depending only on n, such that

$$\lambda_1(L) \leq \sqrt{\gamma_n} \cdot \Delta(L)^{1/n}.$$

Although the value of γ_n is only known for $1 \leq n \leq 8$, for "random lattices" the first minimum, and hence Hemite's constant γ_n, can approximated by appealing to the *Gaussian Heuristic*, which states that for a "random lattice" we have

$$\lambda_1(L) \approx \sqrt{\frac{n}{2 \cdot \pi \cdot e}} \cdot \Delta(L)^{1/n}.$$

The classic result in lattice theory (a.k.a. geometry of numbers), is that of Minkowski, which relates the minimal distance to the volume of the fundamental parallelepiped.

Theorem 5.1 (Minkowski)**.** *Let L be a rank-n lattice and $\mathbb{C} \subset L$ be a convex symmetric body about the origin with volume $\mathsf{Vol}(\mathcal{C}) > 2^n \cdot \Delta(L)$. Then $\mathcal{C}$ contains a non-zero vector $\mathbf{x} \in L$.*

The following immediate corollary is also often referred to as "Minkowski's Theorem".

Corollary 5.2 (Minkowski's Theorem)**.** *For any n-dimensional lattice L we have*

$$\lambda_1(L) \leq \sqrt{n} \cdot \Delta(L)^{1/n}.$$

The *dual* L^* of a lattice L is the set of all vectors $\mathbf{y} \in \mathbb{R}^n$ such that $\mathbf{y} \cdot \mathbf{x}^\mathsf{T} \in \mathbb{Z}$ for all $\mathbf{x} \in L$. Given a basis matrix B of L we can compute the basis matrix B^* of L^* via $B^* = (B^{-1})^\mathsf{T}$. Hence we have $\Delta(L^*) = 1/\Delta(L)$. The first minimum of L and the nth minimum of L^* are linked by the *transference theorem* of Banaszczyk which states that for all n-dimensional lattices we have

$$1 \leq \lambda_1(L) \cdot \lambda_n(L^*) \leq n.$$

Thus a lower bound on $\lambda_1(L)$ can be translated into an upper bound on $\lambda_n(L^*)$ and vice versa, a fact which is used often in the analysis of lattice algorithms.

5.1.4. LLL Algorithm: One could ask, given a lattice L does there exist an orthogonal basis? In general the answer to this last question is no. If one looks at the Gram–Schmidt process in more detail one sees that, even if one starts out with integer vectors, the coefficients $\mu_{i,j}$ almost always end up not being integers. Hence, whilst the Gram–Schmidt basis vectors span the same *vector subspace* as the original basis they do not span the same *lattice* as the original basis. This is because we are not allowed to make a change of basis which consists of non-integer coefficients. However, we could try to make a change of basis so that the new basis is "close" to being orthogonal in that

$$|\mu_{i,j}| \leq \frac{1}{2} \text{ for } 1 \leq j < i \leq n.$$

These considerations led Lenstra, Lenstra and Lovász to define the following notion of reduced basis, called an LLL reduced basis after its inventors.

Definition 5.3. *A basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is called LLL reduced if the associated Gram–Schmidt basis $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*\}$ satisfies*

$$|\mu_{i,j}| \leq \frac{1}{2} \text{ for } 1 \leq j < i \leq n, \tag{7}$$

$$\|\mathbf{b}_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right) \cdot \|\mathbf{b}_{i-1}^*\|^2 \text{ for } 1 < i \leq n. \tag{8}$$

What is truly amazing about an LLL reduced basis is

- An LLL reduced basis can be computed in polynomial time; see below for the method.
- The first vector in the reduced basis is very short, in fact it is close to the shortest non-zero vector in that for all non-zero $\mathbf{x} \in L$ we have
$$\|\mathbf{b}_1\| \leq 2^{(n-1)/2} \cdot \|\mathbf{x}\|.$$
- $\|\mathbf{b}_1\| \leq 2^{n/4} \cdot \Delta^{1/n}$.

The constant $2^{(n-1)/2}$ in the second bullet point above is a worst-case constant, and is called the *approximation factor*. In practice for many lattices of small dimension after one applies the LLL algorithm to obtain an LLL reduced basis, the first vector in the LLL reduced basis is in fact equal to the smallest vector in the lattice. Hence, in such cases the approximation factor is one.

The LLL algorithm works as follows: We keep track of a copy of the current lattice basis B and the associated Gram–Schmidt basis B^*. At any point in time we are examining a fixed column k, where we start with $k = 2$.

- If condition (7) does not hold for $\mu_{k,j}$ with $1 \leq j < k$ then we alter the basis B so that it does.
- If condition (8) does not hold for column k and column $k-1$ we swap columns k and $k-1$ around and decrease the value of k by one (unless k is already equal to two). If condition (8) holds then we increase k by one.

At some point (in fact in polynomial time) we will obtain $k = n$ and the algorithm will terminate. To prove this, one shows that the number of iterations where one decreases k is bounded, and so it is guaranteed that the algorithm will terminate. We will not prove this, but note that clearly if the algorithm terminates it will produce an LLL reduced basis.

For a Gram–Schmidt basis B^* of a basis B we define the Gram–Schmidt Log, $\mathsf{GSL}(B)$, as the vector

$$\mathsf{GSL}(B) = \left(\log\left(\|\mathbf{b}_i^*\|/\Delta(L)^{1/n}\right)\right)_{i=1,\ldots,n}.$$

It is "folklore" that the output of the LLL algorithm produces a basis B whose $\mathsf{GSL}(B)$ when plotted looks like a straight line. The (average) slope η_B of this line can be then computed from $\mathsf{GSL}(B)$ via

$$\eta_B := \frac{12}{(n+1)\cdot n\cdot(n-1)}\cdot\left(\sum_{i=1}^{n} i\cdot\mathsf{GSL}(B)_i\right).$$

The Geometric Series Assumption (GSA) is that the output of the LLL algorithm does indeed behave in this way for a given input basis.

Example: As an example we take the above basis of a two-dimensional lattice in $\mathbb{R}^2$

$$\mathbf{b}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix},\ \mathbf{b}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The associated Gram–Schmidt basis is given by

$$\mathbf{b}_1^* = \begin{pmatrix} 2 \\ 0 \end{pmatrix},\ \mathbf{b}_2^* = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

But this is not a basis of the associated lattice since one cannot pass from $\{\mathbf{b}_1, \mathbf{b}_2\}$ to $\{\mathbf{b}_1^*, \mathbf{b}_2^*\}$ via a unimodular integer transformation.

We now apply the LLL algorithm with $k = 2$ and find that the first condition (7) is satisfied since $\mu_{2,1} = \frac{1}{2}$. However, the second condition (8) is not satisfied because

$$1 = \|\mathbf{b}_2^*\|^2 < \left(\frac{3}{4} - \mu_{2,1}^2\right)\cdot\|\mathbf{b}_1^*\|^2 = \frac{1}{2}\cdot 4 = 2.$$

Hence, we need to swap the two basis vectors around, to obtain the new lattice basis vectors

$$\mathbf{b}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix},\ \mathbf{b}_2 = \begin{pmatrix} 2 \\ 0 \end{pmatrix},$$

with the associated Gram–Schmidt basis vectors

$$\mathbf{b}_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix},\ \mathbf{b}_2^* = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

We now go back to the first condition again. This time we see that we have $\mu_{2,1} = 1$ which violates the first condition. To correct this we subtract $\mathbf{b}_1$ from the vector $\mathbf{b}_2$ so as to obtain $\mu_{2,1} = 0$. We now find the lattice basis is given by

$$\mathbf{b}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix},\ \mathbf{b}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and the Gram–Schmidt basis is then identical to this lattice basis, in that

$$\mathbf{b}_1^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix},\ \mathbf{b}_2^* = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Now we are left to check the second condition again, we find

$$2 = \|\mathbf{b}_2^*\|^2 \geq \left(\frac{3}{4} - \mu_{2,1}^2\right) \cdot \|\mathbf{b}_1^*\|^2 = \frac{3}{4} \cdot 2 = \frac{3}{2}.$$

Hence, both conditions are satisfied and we conclude that

$$\mathbf{b}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \ \mathbf{b}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

is an LLL reduced basis for the lattice L.

5.1.5. Continued Fractions: We end this section by noticing that there is a link between continued fractions and short vectors in lattices. Let $\alpha \in \mathbb{R}$, and define the following sequences, starting with $\alpha_0 = \alpha$, $p_0 = a_0$ and $q_0 = 1$, $p_1 = a_0 \cdot a_1 + 1$ and $q_1 = a_1$,

$$\begin{aligned} a_i &= \lfloor \alpha_i \rfloor, \\ \alpha_{i+1} &= \frac{1}{\alpha_i - a_i}, \\ p_i &= a_i \cdot p_{i-1} + p_{i-2} \text{ for } i \geq 2, \\ q_i &= a_i \cdot q_{i-1} + q_{i-2} \text{ for } i \geq 2. \end{aligned}$$

The integers $a_0, a_1, a_2, \ldots$ are called the continued fraction expansion of α and the fractions

$$\frac{p_i}{q_i}$$

are called the convergents. The denominators of these convergents grow at an exponential rate and the convergent above is a fraction in its lowest terms since one can show $\gcd(p_i, q_i) = 1$ for all values of i. The important result is that if p and q are two integers with

$$\left|\alpha - \frac{p}{q}\right| \leq \frac{1}{2 \cdot q^2}$$

then $\frac{p}{q}$ is a convergent in the continued fraction expansion of α.

A similar effect can be achieved using lattices by applying the LLL algorithm to the lattice generated by the columns of the matrix

$$\begin{pmatrix} 1 & 0 \\ C \cdot \alpha & -C \end{pmatrix},$$

for some constant C. This is because the lattice L contains the "short" vector

$$\begin{pmatrix} q \\ C \cdot (q \cdot \alpha - p) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ C \cdot \alpha & -C \end{pmatrix} \cdot \begin{pmatrix} q \\ p \end{pmatrix}.$$

Therefore, in some sense we can consider the LLL algorithm to be a multi-dimensional generalization of the continued fraction algorithm.

5.2. "Hard" Lattice Problems

There are two basic hard problems associated with a lattice. The first is the *shortest vector problem* and the second is the *closest vector problem*. However, these problems are only hard for large dimensions, since for large dimension the approximation factor of the LLL algorithm, $2^{(n-1)/2}$, becomes too large and the LLL algorithm is no longer able to find a good basis.

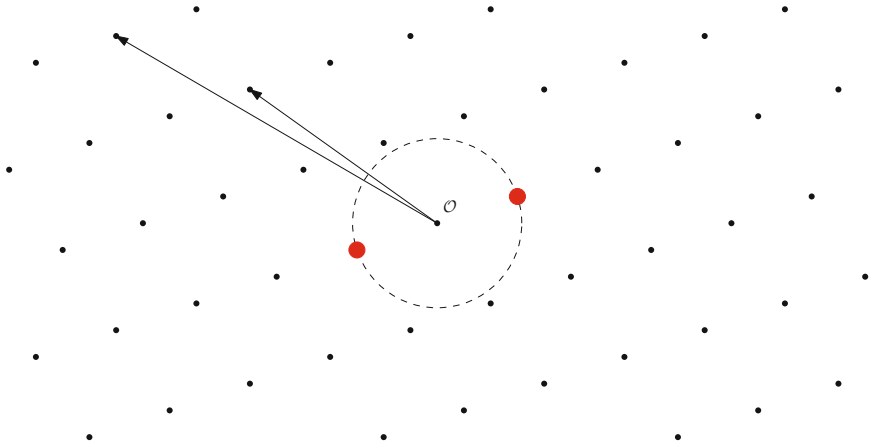


FIGURE 5.3. An SVP solution

5.2.1. Shortest Vector Problem: The simplest, and most famous, hard problem in a lattice is to determine the shortest vector within the lattice. This problem comes in a number of variants:

Definition 5.4 (Shortest Vector Problem)**.** *Given a lattice basis B there are three variants of this problem:*

- *The shortest vector problem* SVP *is to find a non-zero vector* $\mathbf{x}$ *in the lattice L generated by B for which*
$$\|\mathbf{x}\| \leq \|\mathbf{y}\|$$
for all non-zero $\mathbf{y} \in L$, i.e. $\|\mathbf{x}\| = \lambda_1(L)$.
- *The approximate-SVP problem* SVP_γ *is to find a* $\mathbf{x}$ *such that*
$$\|\mathbf{x}\| \leq \gamma \cdot \lambda_1(L),$$
for some "small" constant γ.
- *The γ-unique SVP problem* uSVP_γ *is given a lattice and a constant $\gamma > 1$ such that $\lambda_2(L) > \gamma \cdot \lambda_1(L)$, find a non-zero $\mathbf{x} \in L$ of length $\lambda_1(L)$.*

See Figure 5.3 for an example two-dimensional lattice, the input basis, and the two shortest lattice vectors which an SVP solver should find. Note that a short lattice vector is not unique, since if $\mathbf{x} \in L$ then we also have $-\mathbf{x} \in L$. The LLL algorithm will heuristically solve the SVP, and for large dimension will solve the approximate-SVP problem with a value of γ of $2^{(n-1)/2}$ in the worst case. The γ-unique SVP problem is potentially easier than the others, since we are given more information about the underlying lattice.

5.2.2. Closest Vector Problem: We now present the second most important problem in lattices, namely the closest vector problem. See Figure 5.4 for an example two-dimensional lattice, the input basis, the target vector $\mathbf{x}$ in blue, and the closest lattice vector $\mathbf{y}$ in red.

Definition 5.5 (Closest Vector Problem)**.** *Given a lattice basis B generating a lattice L in n-dimensional real space, and a vector $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{x} \notin L$, there are two variants of this problem:*

- *The closest vector problem* CVP *is to find a lattice vector $\mathbf{y} \in L$ such that*
$$\|\mathbf{x} - \mathbf{y}\| \leq \|\mathbf{x} - \mathbf{z}\|$$
for all $\mathbf{z} \in L$.

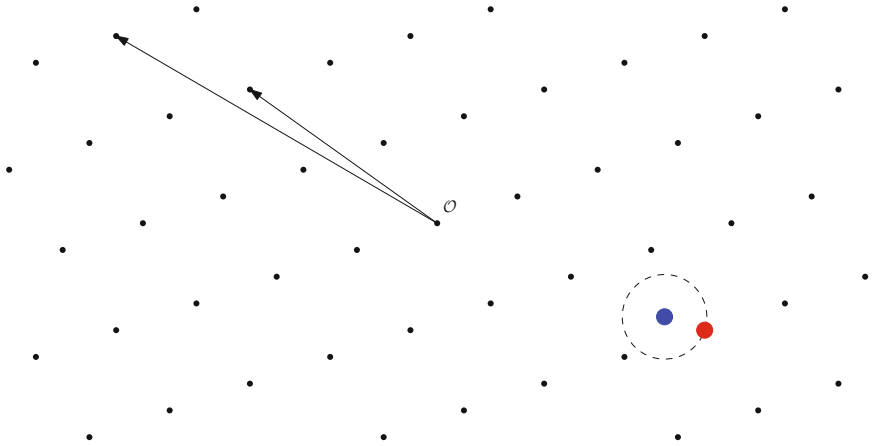


FIGURE 5.4. A CVP solution

- *The approximate-CVP problem* CVP_γ *is to find a* $\mathbf{y}$ *such that*

$$\|\mathbf{x}-\mathbf{y}\| \leq \gamma \cdot \|\mathbf{x}-\mathbf{z}\|$$

for all $\mathbf{z} \in L$*, for some "small" constant* γ*.*

The SVP and CVP problems are related in that, in practice, we can turn an algorithm which finds approximate-SVP solutions into one which finds approximate-CVP solutions as follows. Let $(B, \mathbf{x})$ denote the input to the CVP problem, where without loss of generality we assume that B is an $n \times n$ matrix, i.e. the lattice is full rank and in $\mathbb{R}^n$. We now form the lattice in $\mathbb{R}^{n+1}$ generated by the columns of the matrix

$$B' = \left(\begin{array}{c|c} B & \mathbf{x} \\ \hline \mathbf{0} & C \end{array}\right),$$

where C is some "large" constant. We let $\mathbf{b}'$ denote a short vector in the lattice generated by B', so we have that

$$\mathbf{b}' = \begin{pmatrix} B & \mathbf{x} \\ \mathbf{0} & C \end{pmatrix} \cdot \begin{pmatrix} \mathbf{a} \\ t \end{pmatrix} = \begin{pmatrix} \mathbf{y} + t \cdot \mathbf{x} \\ t \cdot C \end{pmatrix},$$

where $\mathbf{y}$ is in the lattice generated by L, i.e. $\mathbf{y} = B \cdot \mathbf{a}$ for $\mathbf{a} \in \mathbb{Z}^n$. We then have that

$$\|\mathbf{b}'\|^2 = \|\mathbf{y} + t \cdot \mathbf{x}\|^2 + t^2 \cdot C^2$$

Now since $\|\mathbf{b}'\|$ is "small" and C is "large" we expect that $t = \pm 1$. Without loss of generality we can assume that $t = -1$. But this then implies that $\|\mathbf{y} - \mathbf{x}\|^2$ is very small, and so $\mathbf{y}$ is highly likely to be an approximate solution to the original closest vector problem.

5.2.3. Bounded-Distance Decoding Problem: There is a "simpler" problem (meaning we give the solver of the problem more information) associated with the closest vector problem called the Bounded-Distance Decoding problem (or BDD problem). Here we not only give the adversary the lattice and a non-lattice vector, but we also give the adversary a bound on the distance between the lattice and the non-lattice vector. Thus we are giving the adversary more information than in the above two CVP problems; thus the BDD problem is akin to the γ-unique SVP problem.

Definition 5.6 (Bounded-Distance Decoding)**.** *Given a lattice basis* B *generating a lattice* L *in* n*-dimensional real space, a vector* $\mathbf{x} \in \mathbb{R}^n$ *with* $\mathbf{x} \notin L$*, and a real number* λ*. The Bounded-Distance Decoding problem* BDD_λ *is to find a lattice vector* $\mathbf{y} \in L$ *such that*

$$\|\mathbf{x}-\mathbf{y}\| \leq \lambda \cdot \lambda_1(L).$$

If the shortest non-zero vector in the lattice has size $\lambda_1(L)$, then if we have $\lambda < 1/2$ the solution to the BDD problem is guaranteed to be unique. An interesting aspect of the BDD problem is that if the vector $\mathbf{x}$ is known to be really close to the lattice, i.e. λ is much smaller than one would expect for a random value of $\mathbf{x}$, then solving the BDD problem becomes easier.

The BDD problem will be very important later in Chapter 17 so we now discuss how hard BDD is in relation to other more standard lattice problems. The traditional "practical" method of solving BDD_α is to embed the problem into a uSVP_γ problem of a lattice of dimension one larger than the original problem. This is exactly our strategy above for solving CVP. For BDD we can formalize this and show that the two problems are (essentially) equivalent. We present the reduction from BDD_α to uSVP_γ, since we can use lattice reduction algorithms as a uSVP_γ oracle, and hence the reduction in this direction is more practically relevant. We simplify things by assuming that the distance in the BDD_α problem is *exactly* known. Note that a more complex algorithm to that given in the proof can deal with the case where this distance is not known, but is just known to be less than $\alpha \cdot \lambda_1(L)$. In any case, in many examples the distance will be known with a high degree of certainty.

Theorem 5.7. *Given a lattice L via a basis B and a vector $\mathbf{x} \in \mathbb{R}^n$, with $\mathbf{x} \notin L$, such that the minimum distance from $\mathbf{x}$ to a lattice point $\mathsf{dist}(\mathbf{x}, L) = \mu \leq \alpha \cdot \lambda_1(L)$, and an oracle for uSVP_γ for $\gamma = 1/(2 \cdot \alpha)$ then, assuming μ is known, there is an algorithm which will output $\mathbf{y}$ such that $\mathbf{y} \in L$ and $\|\mathbf{y} - \mathbf{x}\| = \mathsf{dist}(\mathbf{x}, L)$.*

Proof. First define the matrix

$$B' = \begin{pmatrix} B & \mathbf{x} \\ \mathbf{0} & \mu \end{pmatrix},$$

and consider the lattice L' generated by the matrix B'. For the target vector $\mathbf{y}$ define $\mathbf{z}$ by $\mathbf{y} = B \cdot \mathbf{z}$, where $\mathbf{z} \in \mathbb{Z}^n$. Consider the vector $\mathbf{y}' \in L'$ defined by

$$\mathbf{y}' = B' \cdot \begin{pmatrix} \mathbf{z} \\ -1 \end{pmatrix} = \begin{pmatrix} B \cdot \mathbf{z} - \mathbf{x} \\ -\mu \end{pmatrix} = \begin{pmatrix} \mathbf{y} - \mathbf{x} \\ -\mu \end{pmatrix}.$$

We have $\|\mathbf{y}'\| = \sqrt{\mu^2 + \mu^2} = \sqrt{2} \cdot \mu$ by assumption that $\mathsf{dist}(\mathbf{x}, L) = \mu$. If we pass the basis B' to our uSVP_γ oracle then this will output a vector $\mathbf{v}'$. We would like $\mathbf{v}' = \mathbf{y}'$ i.e. $\|\mathbf{y}'\| = \lambda_1(L')$ and all other vectors in L' are either a multiple of $\mathbf{y}'$ or have length greater than $\gamma \cdot \|\mathbf{y}'\| = \sqrt{2} \cdot \mu \cdot \gamma$. If this is true we can solve our BDD_α problem by taking the first n coordinates of $\mathbf{y}'$ and adding the vector $\mathbf{x}$ to the result so as to obtain the solution $\mathbf{y}$.

So assume, for sake of contradiction, that $\mathbf{w}'$ is a vector in L' of length less than $\sqrt{2} \cdot \mu \cdot \gamma$ and that $\mathbf{w}'$ is not a multiple of $\mathbf{y}'$. We can write, for $\mathbf{z}_1 \in \mathbb{Z}^n$ and $\beta \in \mathbb{Z}$,

$$\mathbf{w}' = B' \cdot (\mathbf{z}_1, -\beta)^\mathsf{T} = (B \cdot \mathbf{z}_1 - \beta \cdot \mathbf{x}, -\beta \cdot \mu)^\mathsf{T} = (\mathbf{w} - \beta \cdot \mathbf{x}, -\beta \cdot \mu)^\mathsf{T}$$

where $\mathbf{w} = B \cdot \mathbf{z}_1 \in L$. So we have $\sqrt{\|\mathbf{w} - \beta \cdot \mathbf{x}\|^2 + (\beta \cdot \mu)^2} = \|\mathbf{w}'\| < \sqrt{2} \cdot \mu \cdot \gamma$, which implies that $\|\mathbf{w} - \beta \cdot \mathbf{x}\| < \sqrt{2 \cdot \mu^2 \cdot \gamma^2 - \beta^2 \cdot \mu^2}$.

Now consider the vector $\mathbf{w} - \beta \cdot \mathbf{y} \in L$. Since $\mathbf{w}'$ is not a multiple of $\mathbf{y}'$, neither is $\mathbf{w}$ a multiple of $\mathbf{y}$, and so $\mathbf{w} - \beta \cdot \mathbf{y} \neq \mathbf{0}$. We wish to upper bound the length of $\mathbf{w} - \beta \cdot \mathbf{y}$:

$$\begin{aligned} \|\mathbf{w} - \beta \cdot \mathbf{y}\| &= \|(\mathbf{w} - \beta \cdot \mathbf{x}) - \beta \cdot (\mathbf{y} - \mathbf{x})\| \\ &\leq \|\mathbf{w} - \beta \cdot \mathbf{x}\| + \beta \cdot \|\mathbf{y} - \mathbf{x}\| \\ &< \sqrt{2 \cdot \mu^2 \cdot \gamma^2 - \beta^2 \cdot \mu^2} + \beta \cdot \mu. \end{aligned}$$

Now maximizing the right hand side by varying β, we find the maximum is $2 \cdot \gamma \cdot \mu$, which is achieved when $\beta = \gamma$ for real β, and hence we will have $\|\mathbf{w} - \beta \cdot \mathbf{y}\| < 2 \cdot \gamma \cdot \mu$ for integer values of β as well.

We now use the equality $\gamma = 1/(2 \cdot \alpha)$ and the inequality $\mu \leq \alpha \cdot \lambda_1(L)$ to obtain that

$$\|\mathbf{w} - \beta \cdot \mathbf{y}\| < 2 \cdot \left(\frac{1}{2 \cdot \alpha}\right) \cdot (\alpha \cdot \lambda_1(L)) = \lambda_1(L).$$

Thus for all integer values of β we conclude that $\mathbf{w} - \beta \cdot \mathbf{y} \in L$, that $\mathbf{w} - \beta \cdot \mathbf{y} \neq \mathbf{0}$ (since $\mathbf{w}$ is not a multiple of $\mathbf{y}$) and that $\|\mathbf{w} - \beta \cdot \mathbf{y}\| < \lambda_1(L)$ which is a contradiction, since $\lambda_1(L)$ is the length of the smallest non-zero vector in the lattice. □

So if we want to solve the BDD_α problem we convert it into a $\mathsf{uSVP}_{1/2\cdot\alpha}$ problem. We then apply lattice basis reduction to the resulting $\mathsf{uSVP}_{1/2\cdot\alpha}$ problem, and hope the resulting first basis element allows us to solve the original BDD_α problem. Since lattice reduction gets worse as the dimension increases this means as n increases we can only solve BDD problems in which the distance between the target vector and the lattice is very small. Alternatively, if we want BDD to be hard when the distance between the target and the lattice is very small then we need the dimension to be large.

The other approach in the literature for solving BDD_α in practice is to apply Babai's algorithm for solving closest vector problems. Babai's algorithm takes as input a lattice basis B and a non-lattice vector $\mathbf{x}$ and then outputs a lattice vector $\mathbf{w}$ such that the "error" $\mathbf{e} = \mathbf{w} - \mathbf{x}$ lies in the fundamental parallelepiped of the matrix B^*, where B^* is the Gram–Schmidt basis associated with B. In particular we have

$$\|\mathbf{e}\|^2 \leq \frac{1}{4} \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2.$$

If the input basis is reduced we expect this latter quantity to be small, and hence the error vector $\mathbf{e}$ is itself small and represents the distance to the closest lattice vector to $\mathbf{x}$.

5.3. q-ary Lattices

Of importance in one of the systems described later are so-called q-ary lattices. A q-ary lattice L is one such that $q\mathbb{Z}^n \subset L \subset \mathbb{Z}^n$ for some integer q. Note, that all integer lattices are q-ary lattices for a value of q which is an integer multiple of $\Delta(L)$. Our interest will be in special forms of q-ary lattice which are q-ary for a q-value much less than the determinant.

Suppose we are given a matrix $A \in \mathbb{Z}_q^{n\times m}$, with $m \geq n$; we then define the following two m-dimensional q-ary lattices.

$$\Lambda_q(A) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{y} = A^\mathsf{T} \cdot \mathbf{z} \pmod q \text{ for some } \mathbf{z} \in \mathbb{Z}^n\},$$
$$\Lambda_q^\perp(A) = \{\mathbf{y} \in \mathbb{Z}^m : A \cdot \mathbf{y} = 0 \pmod q\}.$$

Suppose we have $\mathbf{y} \in \Lambda_q(A)$ and $\mathbf{y}' \in \Lambda_q^\perp(A)$, then we have $\mathbf{y} = A^\mathsf{T} \cdot \mathbf{z} \pmod q$ and $A \cdot \mathbf{y}' = 0 \pmod q$. This implies that

$$\mathbf{y}^\mathsf{T} \cdot \mathbf{y}' = (\mathbf{z}^\mathsf{T} \cdot A) \cdot \mathbf{y}' = \mathbf{z}^\mathsf{T} \cdot (A \cdot \mathbf{y}') \in q \cdot \mathbb{Z}.$$

Hence, the two lattices are, up to normalization, duals of each other. We have $\Lambda_q(A) = q \cdot \Lambda_q^\perp(A)^*$ and $\Lambda_q^\perp(A) = q \cdot \Lambda_q(A)^*$.

Example: To fix ideas, consider the following example; Let $n = 2$, $m = 3$, $q = 1009$ and set

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 5 & 6 \end{pmatrix}.$$

To define a basis B of $\Lambda_q(A)$ we can take the column-Hermite Normal Form (HNF) of the 3×5 matrix $\left(A^\mathsf{T} \mid q \cdot I_3\right)$ to obtain

$$B = \begin{pmatrix} 1009 & 1 & 336 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The basis of $\Lambda_q^\perp(A)$ is given by

$$B^* = q \cdot ((B^\mathsf{T})^{-1}) = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1009 & 0 \\ -336 & 0 & 1009 \end{pmatrix}.$$

The properties of the above example hold in general; namely if q is prime and m is a bit larger than n then we have $\Delta(\Lambda_q(A)) = q^{m-n}$ and $\Delta(\Lambda_q^\perp(A)) = q^n$. From this, using the Gaussian Heuristic, we find for $A \in \mathbb{Z}_q^{n \times m}$ that we expect

$$\lambda_1(\Lambda_q(A)) \approx \sqrt{\frac{m}{2 \cdot \pi \cdot e}} \cdot q^{(m-n)/m},$$
$$\lambda_1(\Lambda_q^\perp(A)) \approx \sqrt{\frac{m}{2 \cdot \pi \cdot e}} \cdot q^{n/m}.$$

Another lattice-based problem which is of interest in cryptography, and is related to these q-ary lattices, is the Short Integer Solution problem (or SIS problem).

Definition 5.8 (Short Integer Solution)**.** *Given an integer q and vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m \in (\mathbb{Z}/q\mathbb{Z})^n$ the SIS problem is to find a* short *$\mathbf{z} \in \mathbb{Z}^m$ such that*

$$z_1 \cdot \mathbf{a}_1 + \cdots + z_m \cdot \mathbf{a}_m = \mathbf{0} \pmod{q}.$$

Here "short" often means $z_i \in \{-1, 0, 1\}$.

The SIS problem is related to q-ary lattices in the following way. If we set $A = (\mathbf{a}_1, \ldots, \mathbf{a}_m) \in (\mathbb{Z}/q\mathbb{Z})^{n \times m}$ and set

$$\Lambda_q^\perp(A) = \{\mathbf{z} \in \mathbb{Z}^m : A \cdot \mathbf{z} = \mathbf{0} \pmod{q}\},$$

then the SIS problem becomes the shortest vector problem for the lattice $\Lambda^\perp(A)$.

5.4. Coppersmith's Theorem

In this section we examine a standard tool which is used when one applies lattices to attack certain systems. Much of the work in this area is derived from initial work of Coppersmith, which was later simplified by Howgrave-Graham. Coppersmith's contribution was to provide a method to solve the following problem: Given a polynomial of degree d

$$f(x) = f_0 + f_1 \cdot x + \cdots + f_{d-1} \cdot x^{d-1} + x^d$$

over the integers and the side information that there exists a root x_0 modulo N which is small, say $|x_0| < N^{1/d}$, can one efficiently find the small root x_0? The answer is surprisingly yes, and this leads to a number of interesting cryptographic consequences.

The basic idea is to find a polynomial $h(x) \in \mathbb{Z}[x]$ which has the same root modulo N as the target polynomial $f(x)$. This new polynomial $h(x)$ should be small in the sense that the norm of its coefficients,

$$\|h\|^2 = \sum_{i=0}^{\deg(h)} h_i^2$$

should be small. If such an $h(x)$ can be found then we can appeal to the following lemma.

Lemma 5.9. *Let $h(x) \in \mathbb{Z}[x]$ denote a polynomial of degree at most n and let X and N be positive integers. Suppose*

$$\|h(X \cdot x)\| < N/\sqrt{n}$$

then if $|x_0| < X$ satisfies

$$h(x_0) = 0 \pmod{N}$$

then $h(x_0) = 0$ over the integers and not just modulo N.

Thus to solve our original problem we need to find the roots of $h(X)$ over the integers, which can be done in polynomial time via a variety of methods. Now we return to our original polynomial $f(x)$ of degree d and notice that if

$$f(x_0) = 0 \pmod{N}$$

then we also have

$$f(x_0)^k = 0 \pmod{N^k}.$$

Moreover, if we set, for some given value of m,

$$g_{u,v}(x) \leftarrow N^{m-v} \cdot x^u \cdot f(x)^v$$

then

$$g_{u,v}(x_0) = 0 \pmod{N^m}$$

for all $0 \leq u < d$ and $0 \leq v \leq m$. We then fix m and try to find $a_{u,v} \in \mathbb{Z}$ so that

$$h(x) = \sum_{u \geq 0} \left(\sum_{v=0}^{m} a_{u,v} \cdot g_{u,v}(x) \right)$$

satisfies the conditions of the above lemma. In other words we wish to find integer values of $a_{u,v}$, so that the resulting polynomial h satisfies

$$\|h(X \cdot x)\| \leq N^m/\sqrt{d \cdot (m+1)},$$

with

$$h(X \cdot x) = \sum_{u \geq 0} \left(\sum_{v=0}^{m} a_{u,v} \cdot g_{u,v}(X \cdot x) \right).$$

This is a minimization problem which can be solved using lattice basis reduction, as we shall now show in a simple example.

Example: Suppose our polynomial $f(x)$ is given by

$$f(x) = x^2 + a \cdot x + b$$

and we wish to find an x_0 such that

$$f(x_0) = 0 \pmod{N}.$$

We set $m = 2$ in the above construction and compute

$$g_{0,0}(X \cdot x) \leftarrow N^2,$$
$$g_{1,0}(X \cdot x) \leftarrow X \cdot N^2 \cdot x,$$
$$g_{0,1}(X \cdot x) \leftarrow b \cdot N + a \cdot X \cdot N \cdot x + N \cdot X^2 \cdot x^2,$$
$$g_{1,1}(X \cdot x) \leftarrow b \cdot N \cdot X \cdot x + a \cdot N \cdot X^2 \cdot x^2 + N \cdot X^3 \cdot x^3,$$
$$g_{0,2}(X \cdot x) \leftarrow b^2 + 2 \cdot b \cdot a \cdot X \cdot x + (a^2 + 2 \cdot b) \cdot X^2 \cdot x^2 + 2 \cdot a \cdot X^3 \cdot x^3 + X^4 \cdot x^4,$$
$$g_{1,2}(X \cdot x) \leftarrow b^2 \cdot X \cdot x + 2 \cdot b \cdot a \cdot X^2 \cdot x^2 + (a^2 + 2 \cdot b) \cdot X^3 \cdot x^3 + 2 \cdot a \cdot X^4 \cdot x^4 + X^5 \cdot x^5.$$

We are looking for a linear combination of the above six polynomials such that the resulting polynomial has small coefficients. Hence we are led to look for small vectors in the lattice generated

by the columns of the following matrix, where each column represents one of the polynomials above and each row represents a power of x,

$$A = \begin{pmatrix} N^2 & 0 & b \cdot N & 0 & b^2 & 0 \\ 0 & X \cdot N^2 & a \cdot X \cdot N & b \cdot N \cdot X & 2 \cdot a \cdot b \cdot X & X \cdot b^2 \\ 0 & 0 & N \cdot X^2 & a \cdot N \cdot X^2 & (a^2 + 2 \cdot b) \cdot X^2 & 2 \cdot a \cdot b \cdot X^2 \\ 0 & 0 & 0 & N \cdot X^3 & 2 \cdot a \cdot X^3 & (a^2 + 2 \cdot b) \cdot X^3 \\ 0 & 0 & 0 & 0 & X^4 & 2 \cdot a \cdot X^4 \\ 0 & 0 & 0 & 0 & 0 & X^5 \end{pmatrix}.$$

This matrix has determinant equal to

$$\det(A) = N^6 \cdot X^{15},$$

and so applying the LLL algorithm to this matrix we obtain a new lattice basis B. The first vector $\mathbf{b}_1$ in B will satisfy

$$\|\mathbf{b}_1\| \leq 2^{6/4} \cdot \det(A)^{1/6} = 2^{3/2} \cdot N \cdot X^{5/2}.$$

So if we set $\mathbf{b}_1 = A \cdot \mathbf{u}$, with $\mathbf{u} = (u_1, u_2, \ldots, u_6)^\mathsf{T}$, then we form the polynomial

$$h(x) = u_1 \cdot g_{0,0}(x) + u_2 \cdot g_{1,0}(x) + \cdots + u_6 \cdot g_{1,2}(x)$$

then we will have

$$\|h(X \cdot x)\| \leq 2^{3/2} \cdot N \cdot X^{5/2}.$$

To apply Lemma 5.9 we will require that

$$2^{3/2} \cdot N \cdot X^{5/2} < N^2/\sqrt{6}.$$

Hence by determining an integer root of $h(x)$ we will determine the small root x_0 of $f(x)$ modulo N, assuming that

$$|x_0| \leq X = \frac{N^{2/5}}{48^{1/5}}.$$

In particular this will work when $|x_0| \leq N^{0.39}$.

A similar technique can be applied to any polynomial of degree d so as to obtain the following.

Theorem 5.10 (Coppersmith)**.** *Let $f \in \mathbb{Z}[x]$ be a monic polynomial of degree d and N an integer. If there is some root x_0 of f modulo N such that $|x_0| \leq X = N^{1/d-\epsilon}$ then one can find x_0 in time polynomial in $\log N$ and $1/\epsilon$, for fixed values of d.*

Similar considerations apply to polynomials in two variables the analogue of Lemma 5.9 is as follows:

Lemma 5.11. *Let $h(x, y) \in \mathbb{Z}[x, y]$ denote a sum of at most w monomials and suppose*

$$h(x_0, y_0) = 0 \pmod{N^e}$$

for some positive integers N and e where the integers x_0 and y_0 satisfy

$$|x_0| < X \quad \text{and} \quad |y_0| < Y$$

and

$$\|h(X \cdot x, Y \cdot y)\| < N^e/\sqrt{w}.$$

Then $h(x_0, y_0) = 0$ holds over the integers.

However, the analogue of Theorem 5.10 then becomes only a heuristic result.

Chapter Summary

- Lattices are discrete analogues of vector spaces; as such they have a shortest non-zero vector.
- Lattice basis reduction often allows us to find the shortest non-zero vector in a given lattice, thus lattice reduction allows us to solve the shortest vector problem.
- Other lattice problems, such as the CVP and BDD problems, can also be solved if we can find good bases of lattices.
- In small dimensions the LLL algorithm works very well. In larger dimensions, whilst it is fast, it does not produce such a good output lattice.
- The SIS problem is related to the SVP problem in q-ary lattices.
- Coppersmith's Theorem allows us to solve a modular polynomial equation when the solution is known to be small. The method works by building a lattice depending on the polynomial and then applying lattice basis reduction to obtain short vectors within this lattice.

Further Reading

A complete survey of lattice-based methods in cryptography is given in the survey article by Nguyen and Stern. The main paper on Coppersmith's approach is by Coppersmith himself, however the approach was simplified somewhat in the paper of Howgrave-Graham.

D. Coppersmith. *Small solutions to polynomial equations, and low exponent RSA vulnerabilities.* J. Cryptology, **10**, 233–260, 1997.

P. Nguyen and J. Stern. *The two faces of lattices in cryptology.* In CALC '01, LNCS 2146, 146–180, Springer, 2001.

N. Howgrave-Graham. *Finding small roots of univariate modular equations revisited.* In Cryptography and Coding, LNCS 1355, 131–142, Springer, 1997.

CHAPTER 6

Implementation Issues

Chapter Goals

- To show how exponentiation algorithms are implemented.
- To explain how modular arithmetic can be implemented efficiently on large numbers.
- To show how certain tricks can be used to speed up exponentiation operations.
- To show how finite fields of characteristic two can be implemented efficiently.

6.1. Introduction

In this chapter we examine how one actually implements cryptographic operations. We shall mainly be concerned with public key operations since those are the most complex to implement. For example, when we introduce RSA or DSA later we will have to perform a modular exponentiation with respect to a modulus of a thousand or more bits. This means we need to understand the implementation issues involved with both modular arithmetic and exponentiation algorithms.

There is another reason to focus on public key algorithms rather than private key ones: in general public key schemes run much more slowly than symmetric schemes. In fact they can be so slow that their use can make networks and web servers unusable. Hence, efficient implementation is crucial unless one is willing to pay a large performance penalty. The chapter focuses on algorithms used in software; for hardware-based algorithms one often uses different techniques entirely.

6.2. Exponentiation Algorithms

So far in this book, e.g. when discussing primality testing in Chapter 2, we have assumed that computing

$$y = x^d \pmod{n}$$

is an easy operation. We will also need this operation both in RSA and in systems based on discrete logarithms such as ElGamal encryption and DSA. In this section we concentrate on the exponentiation algorithms and assume that we can perform modular arithmetic efficiently. In a later section we shall discuss how to perform modular arithmetic. Firstly note it does not make sense to perform this operation via the sequence

- Compute $r \leftarrow x^d$,
- Compute $y \leftarrow r \pmod{n}$.

To see this, consider

$$123^5 \pmod{511} = 28\,153\,056\,843 \pmod{511} = 359.$$

With this naive method one obtains a huge intermediate result, in our small case above this is

$$28\,153\,056\,843.$$

But in a real 2048-bit exponentiation this intermediate result would be in general $2^{2048} \cdot 2048$ bits long. Such a number requires over 10^{600} gigabytes simply to write down.

6.2.1. Binary Exponentiation: To stop this explosion in the size of any intermediate results we use the fact that we are working modulo n. But even here one needs to be careful; a naive algorithm would compute the above example by computing

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^3 &= x \times x^2 \pmod{511} = 316, \\ x^4 &= x \times x^3 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

This requires four modular multiplications, which seems fine for our small example. But for a general exponentiation by a 2048-bit exponent using this method would require around 2^{2048} modular multiplications. If each such multiplication could be done in under one millionth of a second we would still require around 10^{600} years to perform this operation.

However, it is easy to see that, even in our small example, we can reduce the number of required multiplications by being a little more clever:

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^4 &= x^2 \times x^2 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

Which only requires three modular multiplications rather than the previous four. To understand why we only require three modular multiplications notice that the exponent 5 has binary representation $0b101$ and so

- Has bit length $t = 3$,
- Has Hamming weight $h = 2$.

In the above example we required $1 = (h - 1)$ general multiplications and $2 = (t - 1)$ squarings. This fact holds in general, in that a modular exponentiation can be performed using

- $(h - 1)$ multiplications,
- $(t - 1)$ squarings,

where t is the bit length of the exponent and h is the Hamming weight. The average Hamming weight of an integer is $t/2$ so the number of multiplications and squarings is on average

$$t + t/2 - 1.$$

For a 2048-bit modulus this means that the average number of modular multiplications needed to perform exponentiation by a 2048-bit exponent is at most 4096 and on average 3072.

Algorithm 6.1: Binary exponentiation: Right-to-left variant

```
y ← 1.
while d ≠ 0 do
    if (d mod 2) ≠ 0 then
        y ← (y · x) mod n.
        d ← d − 1.
    d ← d/2.
    x ← (x · x) mod n.
```

The method used to achieve this improvement in performance is called the binary exponentiation method. This is because it works by reading each bit of the binary representation of the exponent in turn, starting with the least significant bit and working up to the most significant bit. Algorithm 6.1 explains the method by computing

$$y = x^d \pmod{n}.$$

The above binary exponentiation algorithm has a number of different names: some authors call it the *square and multiply* algorithm, since it proceeds by a sequence of squarings and multiplications, other authors call it the *Indian exponentiation* algorithm. Algorithm 6.1 is called a *right-to-left* exponentiation algorithm since it processes the bits of d from the least significant bit (the right one) up to the most significant bit (the left one).

6.2.2. Window Exponentiation Methods: Most of the time it is faster to perform a squaring operation than a general multiplication. Hence to reduce time even more one tries to reduce the total number of modular multiplications even further. This is done using *window* techniques which trade off precomputations (i.e. storage) against the time in the main loop.

To understand window methods better we first examine the binary exponentiation method again. But this time instead of a right-to-left variant, we process the exponent from the most significant bit first, thus producing a *left-to-right* binary exponentiation algorithm; see Algorithm 6.2. Again we assume we wish to compute

$$y = x^d \pmod{n}.$$

We first give a notation for the binary representation of the exponent

$$d = \sum_{i=0}^{t} d_i 2^i,$$

where $d_i \in \{0, 1\}$. The algorithm processes a single bit of the exponent on every iteration of the loop. Again the number of squarings is equal to t and the expected number of multiplications is equal to $t/2$.

Algorithm 6.2: Binary exponentiation: Left-to-right variant

$y \leftarrow 1$.
for $i = t$ **downto** 0 **do**
  $y \leftarrow (y \cdot y) \mod n$.
  if $d_i = 1$ **then** $y \leftarrow (y \cdot x) \mod n$.

In a window method we process w bits of the exponent at a time, as in Algorithm 6.3. We first precompute a table

$$x_i = x^i \pmod{n} \text{ for } i = 0, \ldots, 2^w - 1.$$

Then we write our exponent out, but this time taking w bits at a time,

$$d = \sum_{i=0}^{t/w} d_i \cdot 2^{i \cdot w},$$

where $d_i \in \{0, 1, 2, \ldots, 2^w - 1\}$.

It is perhaps easier to illustrate this with an example. Suppose we wish to compute

$$y = x^{215} \pmod{n}$$

with a window width of $w = 3$. We compute the d_i as

$$215 = 3 \cdot 2^6 + 2 \cdot 2^3 + 7.$$

Algorithm 6.3: Window exponentiation method

$y \leftarrow 1$.
for $i = t/w$ **downto** 0 **do**
  for $j = 0$ **to** $w - 1$ **do** $y \leftarrow (y \cdot y) \pmod n$.
  $j \leftarrow d_i$.
  $y \leftarrow (y \cdot x_j) \pmod n$.

Hence, our iteration to compute $x^{215} \pmod n$ computes in order

$$\begin{aligned} y &= 1, \\ y &= y \cdot x^3 = x^3, \\ y &= y^8 = x^{24}, \\ y &= y \cdot x^2 = x^{26}, \\ y &= y^8 = y^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

6.2.3. Sliding Window Method: With a window method as above, we still perform t squarings but the number of multiplications reduces to t/w on average. One can do even better by adopting a sliding window method, where we now encode our exponent as

$$d = \sum_{i=0}^{l} d_i \cdot 2^{e_i}$$

where $d_i \in \{1, 3, 5, \ldots, 2^w - 1\}$ and $e_{i+1} - e_i \geq w$. By choosing only odd values for d_i and having a variable window width we achieve both decreased storage for the precomputed values and improved efficiency. After precomputing $x_i = x^i$ for $i = 1, 3, 5, \ldots, 2^w - 1$, we execute Algorithm 6.4.

Algorithm 6.4: Sliding window exponentiation

$y \leftarrow 1$.
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $y \leftarrow (y \cdot y) \pmod n$.
  $j \leftarrow d_i$.
  $y \leftarrow (y \cdot x_j) \pmod n$.
for $j = 0$ **to** $e_0 - 1$ **do** $y \leftarrow (y \cdot y) \pmod n$.

The number of squarings remains again at t, but now the number of multiplications reduces to l, which is about $t/(w + 1)$ on average. In our example of computing $y = x^{215} \pmod n$ we have

$$215 = 2^7 + 5 \cdot 2^4 + 7,$$

and so we execute the steps

$$\begin{aligned} y &= 1, \\ y &= y \cdot x = x^1, \\ y &= y^8 = x^8, \\ y &= y \cdot x^5 = x^{13}, \\ y &= y^{16} = x^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

6.2.4. Generalizations to Any Group: Notice that all of the above window algorithms apply to exponentiation in any abelian group and not just the integers modulo n. Hence, we can use these algorithms to compute α^d in a finite field or to compute $[d]P$ on an elliptic curve; in the latter case we call this point multiplication rather than exponentiation.

An advantage with elliptic curve variants is that negation comes for free, in that given P it is easy to compute $-P$. This leads to the use of signed binary and signed window methods. We only present the signed window method. We precompute

$$P_i = [i]P \text{ for } i = 1, 3, 5, \ldots, 2^{w-1} - 1,$$

which requires only half the storage of the equivalent sliding window method or one quarter of the storage of the equivalent standard window method. We now write our multiplicand d as

$$d = \sum_{i=0}^{l} d_i \cdot 2^{e_i}$$

where $d_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. The signed sliding window method for elliptic curves is then given by Algorithm 6.5.

Algorithm 6.5: Signed sliding window method

$Q \leftarrow 0$.
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $Q \leftarrow [2]Q$.
  $j \leftarrow d_i$.
  if $j > 0$ **then** $Q \leftarrow Q + P_j$.
  else $Q \leftarrow Q - P_{-j}$.
for $j = 0$ **to** $e_0 - 1$ **do** $Q \leftarrow [2]Q$.

6.3. Special Exponentiation Methods

To speed up public key algorithms even more in practice, various tricks are used, the precise one depending on whether we are performing an operation with a public exponent or a private exponent.

6.3.1. Small Exponents: When we compute $y = x^e \pmod{n}$ where the evaluator does not know the factors of n, but knows the exponent e, we often select e to be very small (this does not seem to create any major attacks), for example $e = 3, 17$ or $65\,537$. The reason for these particular values is that they have small Hamming weight, in fact the smallest possible for a non-trivial exponent, namely two. This means that the binary method, or any other exponentiation algorithm, will

require only one general multiplication, but it will still need k squarings where k is the bit size of the exponent e. For example

$$\begin{aligned} x^3 &= x^2 \times x, \\ x^{17} &= x^{16} \times x, \\ &= (((x^2)^2)^2)^2 \times x. \end{aligned}$$

6.3.2. Knowing p and q: In the case of RSA decryption, or signing, the exponent will be a general and secret 2000-bit number. Hence, we need some way of speeding up the computation. Luckily, since we are considering a private key operation we have access to the prime factors of n,

$$n = p \cdot q.$$

Suppose we wish to compute

$$y = x^d \pmod{n}.$$

We speed up the calculation by first computing y modulo p and q:

$$\begin{aligned} y_p &= x^d \pmod{p} = x^{d \pmod{p-1}} \pmod{p}, \\ y_q &= x^d \pmod{q} = x^{d \pmod{q-1}} \pmod{q}. \end{aligned}$$

Since p and q are 1024-bit numbers, the above calculation requires two exponentiations modulo 1024-bit moduli and 1024-bit exponents. This is faster than a single exponentiation modulo a 2048-bit number with a 2048-bit exponent.

But we now need to recover y from y_p and y_q, which is done using the Chinese Remainder Theorem as follows: We compute $t = p^{-1} \pmod{q}$ and store it with the values p and q. The value y can then be recovered from y_p and y_q via

- $u = (y_q - y_p) \cdot t \pmod{q}$,
- $y = y_p + u \cdot p$.

This is why later on in Chapter 15 we say that when you generate a private key it is best to store p and q even though they are not mathematically needed.

6.3.3. Multi-exponentiation: Sometimes we need to compute

$$r = g^a \cdot y^b \pmod{n}.$$

This can be accomplished by first computing g^a and then y^b and then multiplying the results together. However, often it is easier to perform the two exponentiations simultaneously. There are a number of techniques to accomplish this, using various forms of window techniques etc. But all are essentially based on the following idea, called Shamir's trick.

We first compute the look-up table

$$G_i = g^{i_0} \cdot y^{i_1}$$

where $i = (i_1, i_0)$ is the binary representation of i, for $i = 0, 1, 2, 3$. We then compute an exponent array from the two exponents a and b. This is a two-by-t array, where t is the maximum bit length of a and b. The rows of this array are the binary representation of the exponents a and b. We then let I_j, for $j = 1, \ldots, t$, denote the integers whose binary representation is given by the columns of this array. The exponentiation is then computed by setting $r = 1$ and computing

$$r = r^2 \cdot G_{I_j}$$

for $j = 1$ to t. As an example suppose we wish to compute

$$r = g^{11} \cdot y^7,$$

hence we have $t = 4$. We precompute

$$G_0 = 1,\ G_1 = g,\ G_2 = y,\ G_3 = g \cdot y.$$

Since the binary representation of 11 and 7 is given by 1011 and 111, our exponent array is given by

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

The integers I_j then become

$$I_1 = 1,\ I_2 = 2,\ I_3 = 3,\ I_4 = 3.$$

Hence, the four steps of our algorithm become

$$\begin{aligned} r &= G_1 = g, \\ r &= r^2 \cdot G_2 = g^2 \cdot y, \\ r &= r^2 \cdot G_3 = (g^4 \cdot y^2) \cdot (g \cdot y) = g^5 \cdot y^3, \\ r &= r^2 \cdot G_3 = (g^{10} \cdot y^6) \cdot (g \cdot y) = g^{11} \cdot y^7. \end{aligned}$$

Note that elliptic curve analogues of Shamir's trick and its variants exist, which make use of signed representations for the exponent. We do not give these here, but leave them for the interested reader to investigate.

6.4. Multi-precision Arithmetic

We shall now explain how to perform modular arithmetic on 2048-bit numbers. We show how this is accomplished using modern processors, and then go on to show why naive algorithms are usually replaced with a special technique due to Montgomery.

In a cryptographic application it is common to focus on a fixed length for the integers in use, for example 2048 bits in an RSA/DSA implementation or 256 bits for an ECC implementation. This leads to different programming choices than when we implement a general-purpose multi-precision arithmetic library. For example, we no longer need to worry so much about dynamic memory allocation, and we can now concentrate on particular performance enhancements for the integer sizes we are dealing with.

It is common to represent all integers in little-wordian format. This means that if a large integer is held in memory locations $x_0, x_1, \ldots, x_n$, then x_0 is the least significant word and x_n is the most significant word. For a 64-bit machine and 128-bit numbers we would represent x and y as $[x_0, x_1]$ and $[y_0, y_1]$ where

$$\begin{aligned} x &= x_1 \cdot 2^{64} + x_0, \\ y &= y_1 \cdot 2^{64} + y_0. \end{aligned}$$

6.4.1. Addition: Most modern processors have a carry flag which is set by any overflow from an addition operation. Also most have a special instruction, usually called something like **addc**, which adds two integers together and adds on the contents of the carry flag. So if we wish to add our two 128-bit integers given earlier then we need to compute

$$z = x + y = z_2 \cdot 2^{128} + z_1 \cdot 2^{64} + z_0.$$

The values of z_0, z_1 and z_2 are then computed via

```
z0 <- add   x0,y0
z1 <- addc  x1,y1
z2 <- addc  0,0
```

Note that the value held in z_2 is at most one, so the value of z could be a 129-bit integer. The above technique for adding two 128-bit integers can clearly be scaled to adding integers of any fixed length, and can also be made to work for subtraction of large integers.

6.4.2. Schoolbook Multiplication: We now turn to the next simplest arithmetic operation, after addition and subtraction, namely multiplication. Notice that two 64-bit words multiply together to form a 128-bit result, and so most modern processors have an instruction which will perform this operation.

$$w_1 \cdot w_2 = (High, Low) = (H(w_1 \cdot w_2), L(w_1 \cdot w_2)).$$

When we use schoolbook long multiplication, for our two 128-bit numbers, we obtain something like

$$\begin{array}{rrrr} & & x_1 & x_0 \\ & \times & y_1 & y_0 \\ \hline & & H(x_0 \cdot y_0) & L(x_0 \cdot y_0) \\ & H(x_0 \cdot y_1) & L(x_0 \cdot y_1) & \\ & H(x_1 \cdot y_0) & L(x_1 \cdot y_0) & \\ H(x_1 \cdot y_1) & L(x_1 \cdot y_1) & & \end{array}$$

Then we add up the four rows to get the answer, remembering we need to take care of the carries. This then becomes, for

$$z = x \cdot y,$$

something like the following pseudo-code

```
(z1,z0) <- mul x0,y0
(z3,z2) <- mul x1,y1
(h,l)   <- mul x1,y0
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
(h,l)   <- mul x0,y1
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
```

If n denotes the bit size of the integers we are operating on, the above technique for multiplying large integers together clearly requires $O(n^2)$ bit operations, whilst it requires $O(n)$ bit operations to add or subtract integers. It is a natural question as to whether one can multiply integers faster than $O(n^2)$.

6.4.3. Karatsuba Multiplication: One technique to speed up multiplication is called Karatsuba multiplication. Suppose we have two n-bit integers x and y that we wish to multiply. We write these integers as

$$x = x_0 + 2^{n/2} \cdot x_1,$$
$$y = y_0 + 2^{n/2} \cdot y_2,$$

where $0 \leq x_0, x_1, y_0, y_1 < 2^{n/2}$. We then multiply x and y by computing

$$A \leftarrow x_0 \cdot y_0,$$
$$B \leftarrow (x_0 + x_1) \cdot (y_0 + y_1),$$
$$C \leftarrow x_1 \cdot y_1.$$

The product $x \cdot y$ is then given by

$$\begin{aligned} C \cdot 2^n + (B - A - C) \cdot 2^{n/2} + A &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \\ &= (x_0 + 2^{n/2} \cdot x_1) \cdot (y_0 + 2^{n/2} \cdot y_1) \\ &= x \cdot y. \end{aligned}$$

Hence, to multiply two n-bit numbers we require three $n/2$-bit multiplications, two $n/2$-bit additions and three n-bit additions/subtractions. If we denote the cost of an n-bit multiplication by $M(n)$ and the cost of an n-bit addition/subtraction by $A(n)$, we have

$$M(n) = 3 \cdot M(n/2) + 2 \cdot A(n/2) + 3 \cdot A(n).$$

Now if we make the approximation that $A(n) \approx n$ then

$$M(n) \approx 3 \cdot M(n/2) + 4 \cdot n.$$

If the multiplication of the $n/2$-bit numbers is accomplished in a similar fashion then to obtain the final complexity of multiplication we solve the above recurrence relation to obtain

$$\begin{aligned} M(n) &\approx 9 \cdot n^{\frac{\log(3)}{\log(2)}} \text{ as } n \longrightarrow \infty \\ &= 9 \cdot n^{1.58}. \end{aligned}$$

So we obtain an algorithm with asymptotic complexity $O(n^{1.58})$. Karatsuba multiplication becomes faster than the $O(n^2)$ method for integers of sizes greater than a few hundred bits. However, one can do even better for very large integers since the fastest known multiplication algorithm takes time

$$O(n \cdot \log n \cdot \log \log n).$$

But neither this latter technique nor Karatsuba multiplication are used in many cryptographic applications. The reason for this will become apparent as we discuss integer division.

6.4.4. Division: After having looked at multiplication we are left with the division operation, which is the hardest of all the basic algorithms. After all division is required in order to be able to compute the remainder on division. Given two large integers x and y we wish to be able to compute q and r such that

$$x = q \cdot y + r$$

where $0 \leq r < y$; such an operation is called a Euclidean division. If we write our two integers x and y in the little-wordian format

$$x = (x_0, \ldots, x_n) \text{ and } y = (y_0, \ldots, y_t)$$

where the base for the representation is $b = 2^w$ then the Euclidean division can be performed by Algorithm 6.6. We let $u \ll_w v$ denote a large integer u shifted to the left by v words, in other words the result of multiplying u by b^v. As one can see this is a complex operation, hence one should try to avoid divisions as much as possible.

6.4.5. Montgomery Arithmetic: That division is a complex operation means our cryptographic operations run very slowly if we use standard division operations as above. Virtually all of the public key systems we will consider will make use of arithmetic modulo another number. What we require is the ability to compute remainders (i.e. to perform modular arithmetic) without having to perform any costly division operations. This at first sight may seem a state of affairs which is impossible to reach, but it can be achieved using a special form of arithmetic called Montgomery arithmetic.

Montgomery arithmetic works by using an alternative representation of integers, called the Montgomery representation. Let us fix some notation; we let b denote 2 to the power of the word

Algorithm 6.6: Euclidean division algorithm

```
r ← x.
/* Cope with the trivial case */
if t > n then
    q ← 0.
    return.
q ← 0, s ← 0.
/* Normalize the divisor */
while y_t < b/2 do
    y ← 2 · y, r ← 2 · r, s ← s + 1.
if r_{n+1} ≠ 0 then n ← n + 1.
/* Get the most significant word of the quotient */
while r ≥ (y ≪_w (n − t)) do
    q_{n−t} ← q_{n−t} + 1.
    r ← r − (y ≪_w n − t).
/* Deal with the rest */
for i = n to t + 1 do
    if r_i = y_t then q_{i−t−1} ← b − 1.
    else q_{i−t−1} ← ⌊(r_i · b + r_{i−1})/y_t⌋.
    if t ≠ 0 then h_m ← y_t · b + y_{t−1}.
    else h_m ← y_t · b.
    h ← q_{i−t−1} · h_m.
    if i ≠ 1 then l ← r_i · b^2 + r_{i−1} · b + r_{i−2}.
    else l ← r_i · b^2 + r_{i−1} · b.
    while h > l do
        q_{i−t−1} ← q_{i−t−1} − 1.
        h ← h − h_m.
    r ← r − (q_{i−t−1} · y) ≪_w (i − t − 1).
    if r < 0 then
        r ← r + (y ≪_w i − t − 1).
        q_{i−t−1} ← q_{i−t−1} − 1.
/* Renormalize */
for i = 0 to s − 1 do r ← r/2.
```

size of our computer, for example $b = 2^{64}$. To perform arithmetic modulo N we choose an integer R which satisfies

$$R = b^t > N,$$

for some integer value of t. Now instead of holding the value of the integer x in memory, we instead hold the value

$$x_R \leftarrow x \cdot R \pmod{N}.$$

Again this is usually held in a little-wordian format. The value x_R is called the Montgomery representation of the integer $x \pmod{N}$. Adding two elements in Montgomery representation is

easy; see Algorithm 6.7. If

$$z = x + y \pmod{N}$$

then given $x \cdot R \pmod{N}$ and $y \cdot R \pmod{N}$ we need to compute $z \cdot R \pmod{N}$.

Algorithm 6.7: Addition in Montgomery representation

$z_R \leftarrow x_R + y_R$.
if $z_R \geq N$ **then** $z_R \leftarrow z_R - N$.

Example: Let us take a simple example with

$$N = 18\,443\,759\,776\,216\,676\,723,$$
$$b = R = 2^{64} = 18\,446\,744\,073\,709\,551\,616.$$

The following is the map from the normal to Montgomery representation of the integers $1, 2$ and 3.

$$1 \longrightarrow 1 \cdot R \pmod{N} = 2\,984\,297\,492\,874\,893,$$
$$2 \longrightarrow 2 \cdot R \pmod{N} = 5\,968\,594\,985\,749\,786,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 8\,952\,892\,478\,624\,679.$$

We can now verify that addition works since we have in the standard representation

$$1 + 2 = 3$$

whilst this is mirrored in the Montgomery representation as

$$2\,984\,297\,492\,874\,893 \; + \; 5\,968\,594\,985\,749\,786 = 8\,952\,892\,478\,624\,679 \pmod{N}.$$

Montgomery Reduction: Now we look at multiplication in Montgomery arithmetic. If we simply multiply two elements in Montgomery representation we will obtain

$$(x \cdot R) \cdot (y \cdot R) = x \cdot y \cdot R^2 \pmod{N}$$

but we want $x \cdot y \cdot R \pmod{N}$. Hence, we need to divide the result of the standard multiplication by R. Since R is a power of 2 we hope this should be easy. The process of computing

$$z = y/R \pmod{N}$$

given y and the earlier choice of R, is called Montgomery reduction. We first precompute the integer $q = 1/N \pmod{R}$, which is simple to perform with no divisions using the binary Euclidean algorithm. Then, performing a Montgomery reduction is done using Algorithm 6.8.

Algorithm 6.8: Montgomery reduction

$u \leftarrow (-y \cdot q) \mod R$.
$z \leftarrow (y + u \cdot N)/R$.
if $z \geq N$ **then** $z \leftarrow z - N$.

Note that the reduction modulo R in the first line is easy: we compute $y \cdot q$ using standard algorithms, the reduction modulo R being achieved by truncating the result. This latter trick works since R is a power of b. The division by R in the second line can also be simply achieved: since $y + u \cdot N = 0 \pmod{R}$, we simply shift the result to the right by t words, again since $R = b^t$.

Example: As an example we again take

$$N = 18\,443\,759\,776\,216\,676\,723,$$
$$R = b = 2^{64} = 18\,446\,744\,073\,709\,551\,616.$$

We wish to compute $2 \cdot 3$ in Montgomery representation. Recall

$$2 \longrightarrow 2 \cdot R \pmod{N} = 5\,968\,594\,985\,749\,786 = x,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 8\,952\,892\,478\,624\,679 = y.$$

We then compute, using a standard multiplication algorithm, that

$$w = x \cdot y = 53\,436\,189\,155\,876\,232\,216\,612\,898\,568\,694 = 2 \cdot 3 \cdot R^2.$$

We now need to pass this value of w into our technique for Montgomery reduction, so as to find the Montgomery representation of $x \cdot y$. We find

$$w = 53\,436\,189\,155\,876\,232\,216\,612\,898\,568\,694,$$
$$q = (1/N) \pmod{R} = 14\,241\,249\,658\,089\,591\,739,$$
$$u = -w \cdot q \pmod{R} = 17\,905\,784\,957\,249\,358,$$
$$z = (w + u \cdot N)/R = 17\,905\,784\,957\,249\,358.$$

So the multiplication of x and y in Montgomery arithmetic should be

$$17\,905\,784\,957\,249\,358.$$

We can check that this is the correct value by computing

$$6 \cdot R \pmod{N} = 17\,905\,784\,957\,249\,358.$$

Hence, we see that Montgomery arithmetic allows us to add and multiply integers modulo an integer N without the need for costly division algorithms.

Optimized Montgomery Reduction: Our above method for Montgomery reduction requires two full multi-precision multiplications. So to multiply two numbers in Montgomery arithmetic we require three full multi-precision multiplications. If we are multiplying 2048-bit numbers, this means the intermediate results can grow to be 4096-bit numbers. We would like to do better, and we can.

Suppose y is given in little-wordian format

$$y = (y_0, y_1, \ldots, y_{2t-2}, y_{2t-1}).$$

Then a better way to perform Montgomery reduction is to first precompute $N' = -1/N \pmod{b}$, which is easy and only requires operations on word-sized quantities, and then to execute Algorithm 6.9.

Algorithm 6.9: Word-oriented Montgomery reduction

$z \leftarrow y$.
for $i = 0$ **to** $t - 1$ **do**
  $u \leftarrow (z_i \cdot N') \mod b$.
  $z \leftarrow z + u \cdot N \cdot b^i$.
$z \leftarrow z/R$.
if $z \geq N$ **then** $z \leftarrow z - N$.

Note that since we are reducing modulo b in the first line of the for loop we can execute this initial multiplication using a simple word multiplication algorithm. The second step of the for loop

requires a shift by one word (to multiply by b) and a single *word* $\times$ *bigint* multiply. Hence, we have reduced the need for large intermediate results in the Montgomery reduction step.

Montgomery Multiplication: We can also interleave the multiplication with the reduction to perform a single loop to produce

$$Z = X \cdot Y/R \pmod{N}.$$

So if $X = x \cdot R$ and $Y = y \cdot R$ this will produce

$$Z = (x \cdot y) \cdot R.$$

This procedure is called Montgomery multiplication and allows us to perform a multiplication in Montgomery arithmetic without the need for larger integers, as in Algorithm 6.10. Whilst Montgomery multiplication has complexity $O(n^2)$ as opposed to the $O(n^{1.58})$ of Karatsuba multiplication, it is still preferable to use Montgomery arithmetic since it deals more efficiently with modular arithmetic.

Algorithm 6.10: Montgomery multiplication

$Z \leftarrow 0$.
for $i = 0$ **to** $t - 1$ **do**
  $u \leftarrow ((z_0 + X_i \cdot Y_0) \cdot N') \mod b$.
  $Z \leftarrow (Z + X_i \cdot Y + u \cdot N)/b$.
if $Z \geq N$ **then** $Z \leftarrow Z - N$.

6.5. Finite Field Arithmetic

Apart from the integers modulo a large prime p the other type of finite field used in cryptography are those of characteristic two. These occur in the AES algorithm and in certain elliptic curve systems. In AES the field is so small that one can use look-up tables or special circuits to perform the basic arithmetic tasks, so in this section we shall concentrate on fields of large degree over $\mathbb{F}_2$, such as those used for elliptic curves. In addition we shall concern ourselves with software implementations only. Fields of characteristic two can have special types of hardware implementations based on optimal normal bases, but we shall not concern ourselves with these.

Recall that to define a finite field of characteristic two we first pick an irreducible polynomial $f(x)$ over $\mathbb{F}_2$ of degree n. The field is defined to be

$$\mathbb{F}_{2^n} = \mathbb{F}_2[x]/f(x),$$

i.e. we look at binary polynomials modulo $f(x)$. Elements of this field are usually represented as bit strings, which represent a binary polynomial. For example the bit string

$$101010111$$

represents the polynomial

$$x^8 + x^6 + x^4 + x^2 + x + 1.$$

Addition and subtraction of elements in $\mathbb{F}_{2^n}$ is accomplished by simply performing a bit-wise exclusive-or, written $\oplus$, between the two bitstrings. Hence, the difficult tasks are multiplication and division.

6.5.1. Characteristic-Two Field Division: It turns out that division, although slower than multiplication, is easier to describe, so we start with division. To compute α/β, where $\alpha, \beta \in \mathbb{F}_{2^n}$, we first compute β^{-1} and then perform the multiplication $\alpha \cdot \beta^{-1}$. So division is reduced to multiplication and the computation of β^{-1}. One way of computing β^{-1} is to use Lagrange's Theorem which tells us, for $\beta \neq 0$, that we have

$$\beta^{2^n-1} = 1.$$

But this means that

$$\beta \cdot \beta^{2^n-2} = 1,$$

or in other words

$$\beta^{-1} = \beta^{2^n-2} = \beta^{2\cdot(2^{n-1}-1)}.$$

Another way of computing β^{-1} is to use the binary Euclidean algorithm. We take the polynomial $a = f$ and the polynomial b which represents β and then perform Algorithm 6.11, which is a version of the binary Euclidean algorithm, where $\text{lsb}(b)$ refers to the least significant bit of b (in other words the coefficient of x^0).

Algorithm 6.11: Inversion of $b(x)$ modulo $f(x)$

$B \leftarrow 0$, $D \leftarrow 1$.
/* At least one of a and b will have a constant term on every execution of the loop */
while $a \neq 0$ **do**
  while $\text{lsb}(a) = 0$ **do**
    $a \leftarrow a \gg 1$.
    if $\text{lsb}(B) \neq 0$ **then** $B \leftarrow B \oplus f$.
    $B \leftarrow B \gg 1$.
  while $\text{lsb}(b) = 0$ **do**
    $b \leftarrow b \gg 1$.
    if $\text{lsb}(D) \neq 0$ **then** $D \leftarrow D \oplus f$.
    $D \leftarrow D \gg 1$.
  /* Now both a and b have a constant term */
  if $\deg(a) \geq \deg(b)$ **then**
    $a \leftarrow a \oplus b$.
    $B \leftarrow B \oplus D$.
  else
    $b \leftarrow a \oplus b$.
    $D \leftarrow D \oplus B$.
return D.

6.5.2. Characteristic-Two Field Multiplication: We now turn to the multiplication operation. Unlike the case of integers modulo N or p, where we use a special method of Montgomery arithmetic, in characteristic two we have the opportunity to choose a polynomial $f(x)$ which has "nice" properties. Any irreducible polynomial of degree n can be used to implement the finite field $\mathbb{F}_{2^n}$, we just need to select the best one.

Almost always one chooses a value of $f(x)$ which is either a trinomial

$$f(x) = x^n + x^k + 1$$

or a pentanomial

$$f(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1.$$

It turns out that for all fields of degree less than 10 000 we can always find such a trinomial or pentanomial to make the multiplication operation very efficient. Table 6.1 at the end of this chapter gives a list for all values of n between 2 and 500 of an example pentanomial or trinomial which defines the field $\mathbb{F}_{2^n}$. In all cases where a trinomial exists we give one, otherwise we present a pentanomial.

Now to perform a multiplication of α by β we first multiply the polynomials representing α and β together to form a polynomial $\gamma(x)$ of degree at most $2 \cdot n - 2$. Then we reduce this polynomial by taking the remainder on division by the polynomial $f(x)$.

We show how this remainder on division is efficiently performed for trinomials, and leave the pentanomial case for the reader. We write

$$\gamma(x) = \gamma_1(x) \cdot x^n + \gamma_0(x).$$

Hence, $\deg(\gamma_1(x)), \deg(\gamma_0(x)) \leq n - 1$. We can then write, as $x^n = x^k + 1 \pmod{x^n + x^k + 1}$,

$$\gamma(x) \pmod{f(x)} = \gamma_0(x) + (x^k + 1) \cdot \gamma_1(x).$$

The right-hand side of this equation can be computed from the bit operations

$$\delta = \gamma_0 \oplus \gamma_1 \oplus (\gamma_1 \ll k).$$

Now δ, as a polynomial, will have degree at most $n - 1 + k$. So we need to carry out this procedure again by first writing

$$\delta(x) = \delta_1(x) \cdot x^n + \delta_0(x),$$

where $\deg(\delta_0(x)) \leq n - 1$ and $\deg(\delta_1(x)) \leq k - 1$. We then compute as before that γ is equivalent to

$$\delta_0 \oplus \delta_1 \oplus (\delta_1 \ll k).$$

This latter polynomial will have degree $\max(n - 1, 2k - 1)$, so if we select our trinomial so that

$$k \leq n/2,$$

then Algorithm 6.12 will perform our division-with-remainder step. Let g denote the polynomial of degree $2 \cdot n - 2$ that we wish to reduce modulo f, where we assume a bit representation for these polynomials.

Algorithm 6.12: Reduction of g by a trinomial

$g_1 \leftarrow g \gg n$.
$g_0 \leftarrow g[n - 1 \ldots 0]$.
$g \leftarrow g_0 \oplus g_1 \oplus (g_1 \ll k)$.
$g_1 \leftarrow g \gg n$.
$g_0 \leftarrow g[n - 1 \ldots 0]$.
$g \leftarrow g_0 \oplus g_1 \oplus (g_1 \ll k)$.

So to complete our description of how to multiply elements in $\mathbb{F}_{2^n}$ we need to explain how to perform the multiplication of two binary polynomials of large degree $n - 1$. Again one can use a naive multiplication algorithm. Often however one uses a look-up table for multiplication of polynomials of degree less than eight, i.e. for operands which fit into one byte. Then multiplication of larger-degree polynomials is reduced to multiplication of polynomials of degree less than eight by using a variant of the standard long multiplication algorithm from school. This algorithm will have complexity $O(n^2)$, where n is the degree of the polynomials involved.

Suppose we have a routine which uses a look-up table to multiply two binary polynomials of degree less than eight, returning a binary polynomial of degree less than sixteen. This function we denote by $\mathsf{MultTab}(a, b)$ where a and b are 8-bit integers representing the input polynomials. To perform a multiplication of two n-bit polynomials represented by two n-bit integers x and y we

perform Algorithm 6.13, where $y \gg 8$ (resp. $y \ll 8$) represents shifting to the rightmost (resp. leftmost) by 8 bits.

Algorithm 6.13: Multiplication of two n-bit polynomials x and y over $\mathbb{F}_2$

```
i ← 0, a ← 0.
while x ≠ 0 do
    u ← y, j ← 0.
    while u ≠ 0 do
        w ← MultTab(x&255, u&255).
        w ← w ≪ (8 · (i + j)).
        a ← a ⊕ w.
        u ← u ≫ 8.
        j ← j + 1.
    x ← x ≫ 8.
    i ← i + 1.
return a.
```

6.5.3. Karatsuba Multiplication: Just as with integer multiplication one can use a divide-and-conquer technique based on Karatsuba multiplication, which again will have a complexity of $O(n^{1.58})$. Suppose the two polynomials we wish to multiply are given by

$$a = a_0 + a_1 \cdot x^{n/2},$$
$$b = b_0 + b_1 \cdot x^{n/2},$$

where a_0, a_1, b_0, b_1 are polynomials of degree less than $n/2$. We then multiply a and b by computing

$$A \leftarrow a_0 \cdot b_0,$$
$$B \leftarrow (a_0 + a_1) \cdot (b_0 + b_1),$$
$$C \leftarrow a_1 \cdot b_1.$$

The product $a \cdot b$ is then given by

$$\begin{aligned} C \cdot x^n + (B - A - C) \cdot x^{n/2} + A &= a_1 \cdot b_1 \cdot x^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot x^{n/2} + a_0 \cdot b_0 \\ &= (a_0 + a_1 \cdot x^{n/2}) \cdot (b_0 + b_1 \cdot x^{n/2}) \\ &= a \cdot b. \end{aligned}$$

Again to multiply a_0 and b_0 etc. we use the Karatsuba multiplication method recursively. Once we reduce to the case of multiplying two polynomials of degree less than eight we resort to using our look-up table to perform the polynomial multiplication. Unlike the integer case we now find that Karatsuba multiplication is more efficient than the schoolbook method even for polynomials of quite small degree, say $n \approx 32$.

6.5.4. Squaring in Characteristic Two: One should note that squaring polynomials in fields of characteristic two is particularly easy. Suppose we have a polynomial

$$a = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3,$$

where $a_i = 0$ or 1. Then to square a we simply "thin out" the coefficients, as $2 = 0 \pmod 2$, as follows:

$$a^2 = a_0 + a_1 \cdot x^2 + a_2 \cdot x^4 + a_3 \cdot x^6.$$

This means that squaring an element in a finite field of characteristic two is very fast compared with a multiplication operation.

Chapter Summary

- Modular exponentiation, or exponentiation in any group, can be computed using the binary exponentiation method. Often it is more efficient to use a window based method, or to use a signed-exponentiation method in the case of elliptic curves.
- There are some special optimizations in various cases. In the case of a known exponent we hope to choose one which is both small and has very low Hamming weight. For exponentiation by a private exponent we use knowledge of the prime factorization of the modulus and the Chinese Remainder Theorem.
- Simultaneous exponentiation is often more efficient than performing two single exponentiations and then combining the result.
- Modular arithmetic is usually implemented using the technique of Montgomery representation. This allows us to avoid costly division operations by replacing the division with simple shift operations. This however is at the expense of using a non-standard representation for the numbers.
- Finite fields of characteristic two can also be implemented efficiently, but now the modular reduction operation can be made simple by choosing a special polynomial $f(x)$. Inversion is also particularly simple using a variant of the binary Euclidean algorithm, although often inversion is still three to ten times slower than multiplication.

Further Reading

The standard reference work for the type of algorithms considered in this chapter is Volume 2 of Knuth. A more gentle introduction can be found in the book by Bach and Shallit, whilst for more algorithms one should consult the book by Cohen. The first chapter of Cohen gives a number of lessons learnt in the development of the PARI/GP calculator which can be useful, whilst Bach and Shallit provides an extensive bibliography and associated commentary.

E. Bach and S. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*. MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1993.

D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1975.

TABLE 6.1. Trinomials and pentanomials

n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$
2	1	3	1	4	1
5	2	6	1	7	1
8	7,3,2	9	1	10	3
11	2	12	3	13	4,3,1
14	5	15	1	16	5,3,1
17	3	18	3	19	5,2,1
20	3	21	2	22	1
23	5	24	8,3,2	25	3
26	4,3,1	27	5,2,1	28	1
29	2	30	1	31	3
32	7,3,2	33	10	34	7
35	2	36	9	37	6,4,1
38	6,5,1	39	4	40	5,4,3
41	3	42	7	43	6,4,3
44	5	45	4,3,1	46	1
47	5	48	11,5,1	49	9
50	4,3,2	51	6,3,1	52	3
53	6,2,1	54	9	55	7
56	7,4,2	57	4	58	19
59	7,4,2	60	1	61	5,2,1
62	29	63	1	64	11,2,1
65	32	66	3	67	5,2,1
68	33	69	6,5,2	70	37,34,33
71	35	72	36,35,33	73	42
74	35	75	35,34,32	76	38,33,32
77	38,33,32	78	41,37,32	79	40,36,32
80	45,39,32	81	35	82	43,35,32
83	39,33,32	84	35	85	35,34,32
86	49,39,32	87	46,34,32	88	45,35,32
89	38	90	35,34,32	91	41,33,32
92	37,33,32	93	35,34,32	94	43,33,32
95	41,33,32	96	57,38,32	97	33
98	63,35,32	99	42,33,32	100	37
101	40,34,32	102	37	103	72
104	43,33,32	105	37	106	73,33,32
107	54,33,32	108	33	109	34,33,32
110	33	111	49	112	73,51,32
113	37,33,32	114	69,33,32	115	53,33,32
116	48,33,32	117	78,33,32	118	33

119	38	120	41,35,32	121	35,34,32
122	39,34,32	123	42,33,32	124	37
125	79,33,32	126	49	127	63
128	55,33,32	129	46	130	61,33,32
131	43,33,32	132	44,33,32	133	46,33,32
134	57	135	39,33,32	136	35,33,32
137	35	138	57,33,32	139	38,33,32
140	45	141	85,35,32	142	71,33,32
143	36,33,32	144	59,33,32	145	52
146	71	147	49	148	61,33,32
149	64,34,32	150	53	151	39
152	35,33,32	153	71,33,32	154	109,33,32
155	62	156	57	157	47,33,32
158	76,33,32	159	34	160	79,33,32
161	39	162	63	163	48,34,32
164	42,33,32	165	35,33,32	166	37
167	35	168	134,33,32	169	34
170	105,35,32	171	125,34,32	172	81
173	71,33,32	174	57	175	57
176	79,37,32	177	88	178	87
179	80,33,32	180	33	181	46,33,32
182	81	183	56	184	121,39,32
185	41	186	79	187	37,33,32
188	46,33,32	189	37,34,32	190	47,33,32
191	51	192	147,33,32	193	73
194	87	195	50,34,32	196	33
197	38,33,32	198	65	199	34
200	57,35,32	201	59	202	55
203	68,33,32	204	99	205	94,33,32
206	37,33,32	207	43	208	119,34,32
209	45	210	49,35,32	211	175,33,32
212	105	213	75,33,32	214	73
215	51	216	115,34,32	217	45
218	71	219	54,33,32	220	33
221	63,33,32	222	102,33,32	223	33
224	39,33,32	225	32	226	59,34,32
227	81,33,32	228	113	229	64,35,32
230	50,33,32	231	34	232	191,33,32
233	74	234	103	235	34,33,32
236	50,33,32	237	80,34,32	238	73
239	36	240	177,35,32	241	70
242	95	243	143,34,32	244	111
245	87,33,32	246	62,33,32	247	82

248	155,33,32	249	35	250	103
251	130,33,32	252	33	253	46
254	85,33,32	255	52	256	91,33,32
257	41	258	71	259	113,33,32
260	35	261	89,34,32	262	86,33,32
263	93	264	179,33,32	265	42
266	47	267	42,33,32	268	61
269	207,33,32	270	53	271	58
272	165,35,32	273	53	274	67
275	81,33,32	276	63	277	91,33,32
278	70,33,32	279	38	280	242,33,32
281	93	282	35	283	53,33,32
284	53	285	50,33,32	286	69
287	71	288	111,33,32	289	36
290	81,33,32	291	168,33,32	292	37
293	94,33,32	294	33	295	48
296	87,33,32	297	83	298	61,33,32
299	147,33,32	300	45	301	83,33,32
302	41	303	36,33,32	304	203,33,32
305	102	306	66,33,32	307	46,33,32
308	40,33,32	309	107,33,32	310	93
311	78,33,32	312	87,33,32	313	79
314	79,33,32	315	132,33,32	316	63
317	36,34,32	318	45	319	36
320	135,34,32	321	41	322	67
323	56,33,32	324	51	325	46,33,32
326	65,33,32	327	34	328	195,37,32
329	50	330	99	331	172,33,32
332	89	333	43,34,32	334	43,33,32
335	113,33,32	336	267,33,32	337	55
338	86,35,32	339	72,33,32	340	45
341	126,33,32	342	125	343	75
344	135,34,32	345	37	346	63
347	56,33,32	348	103	349	182,34,32
350	53	351	34	352	147,34,32
353	69	354	99	355	43,33,32
356	112,33,32	357	76,34,32	358	57
359	68	360	323,33,32	361	56,33,32
362	63	363	74,33,32	364	67
365	303,33,32	366	38,33,32	367	171
368	283,34,32	369	91	370	139
371	116,33,32	372	111	373	299,33,32
374	42,33,32	375	64	376	227,33,32

377	41	378	43	379	44,33,32
380	47	381	107,34,32	382	81
383	90	384	295,34,32	385	51
386	83	387	162,33,32	388	159
389	275,33,32	390	49	391	37,33,32
392	71,33,32	393	62	394	135
395	301,33,32	396	51	397	161,34,32
398	122,33,32	399	49	400	191,33,32
401	152	402	171	403	79,33,32
404	65	405	182,33,32	406	141
407	71	408	267,33,32	409	87
410	87,33,32	411	122,33,32	412	147
413	199,33,32	414	53	415	102
416	287,38,32	417	107	418	199
419	200,33,32	420	45	421	191,33,32
422	149	423	104,33,32	424	213,34,32
425	42	426	63	427	62,33,32
428	105	429	83,33,32	430	62,33,32
431	120	432	287,34,32	433	33
434	55,33,32	435	236,33,32	436	165
437	40,34,32	438	65	439	49
440	63,33,32	441	35	442	119,33,32
443	221,33,32	444	81	445	146,33,32
446	105	447	73	448	83,33,32
449	134	450	47	451	406,33,32
452	97,33,32	453	87,33,32	454	128,33,32
455	38	456	67,34,32	457	61
458	203	459	68,33,32	460	61
461	194,35,32	462	73	463	93
464	143,33,32	465	59	466	143,33,32
467	156,33,32	468	33	469	116,34,32
470	149	471	119	472	47,33,32
473	200	474	191	475	134,33,32
476	129	477	150,33,32	478	121
479	104	480	169,35,32	481	138
482	48,35,32	483	288,33,32	484	105
485	267,33,32	486	81	487	94
488	79,33,32	489	83	490	219
491	61,33,32	492	50,33,32	493	266,33,32
494	137	495	76	496	43,33,32
497	78	498	155	499	40,33,32
500	75				

Part 2

Historical Ciphers

In this part we discuss some historical ciphers; those who are interested in pressing on with modern cryptography should jump straight to Part 3. However, discussing the construction of historical ciphers and how they were broken enables one to get a view of how modern cryptosystems came to be designed as they are. For example, modern block ciphers are built out of two key primitives, substitution and permutation, both of which occur in the construction of historical ciphers.

Encryption of most data today is accomplished using fast block and stream ciphers. These are examples of symmetric encryption algorithms. In addition all historical, i.e. pre-1960, ciphers are symmetric in nature and share some design principles with modern ciphers. The main drawback of symmetric ciphers is that they give rise to the problem of how to distribute the secret keys, a problem which resulted in the Allied breaks of Enigma and Lorenz during World War II, which we discuss in this part.

CHAPTER 7

Historical Ciphers

Chapter Goals

- To explain a number of historical ciphers, such as the Caesar cipher and the substitution cipher.
- To show how these historical ciphers can be broken because they do not hide the underlying statistics of the plaintext.
- To introduce the concepts of substitution and permutation as basic cipher components.
- To introduce a number of attack techniques, such as chosen plaintext attacks.

7.1. Introduction

An encryption algorithm, or cipher, is a means of transforming plaintext into ciphertext under the control of a secret key. This process is called encryption or encipherment. We write

$$c = e_k(m),$$

where

- m is the plaintext,
- e is the cipher function,
- k is the secret key,
- c is the ciphertext.

The reverse process is called decryption or decipherment, and we write

$$m = d_k(c).$$

Note that the encryption and decryption algorithms e, d are public: the secrecy of m given c depends totally on the secrecy of k.

The above process requires that each party needs access to the secret key. The key needs to be known to both sides, but needs to be kept secret. Encryption algorithms which have this property are called *symmetric cryptosystems* or secret key cryptosystems. There is a form of cryptography which uses two different types of key; one is publicly available and used for encryption whilst the other is private and used for decryption. These latter types of cryptosystems are called *asymmetric cryptosystems* or *public key cryptosystems*, and we shall return to them in a later chapter.

Usually in cryptography the communicating parties are denoted by A and B. However, often one uses the more user-friendly names of Alice and Bob. But you should not assume that the parties are necessarily human; we could be describing a communication being carried out between two autonomous machines. The eavesdropper, bad girl, adversary or attacker is usually given the name Eve.

In this chapter we shall present some historical ciphers which were used in the pre-computer age to encrypt data. We shall show that these ciphers are easy to break as soon as one understands the statistics of the underlying language, in our case English. In Chapter 9 we shall study this

relationship between how easy the cipher is to break and the statistical distribution of the underlying plaintext in more detail.

Letter	Freq. (%)	Letter	Freq. (%)
A	8.2	N	6.7
B	1.5	O	7.5
C	2.8	P	1.9
D	4.2	Q	0.1
E	12.7	R	6.0
F	2.2	S	6.3
G	2.0	T	9.0
H	6.1	U	2.8
I	7.0	V	1.0
J	0.1	W	2.4
K	0.8	X	0.1
L	4.0	Y	2.0
M	2.4	Z	0.1

TABLE 7.1. English letter frequencies

FIGURE 7.1. English letter frequencies

The distribution of English letter frequencies is described in Table 7.1, or graphically in Figure 7.1. As one can see, the most common letters are E and T. It often helps to know second-order statistics about the underlying language, such as which are the most common sequences of two or three letters, called bigrams and trigrams. The most common bigrams in English are given by Table 7.2, with the associated approximate frequencies. The most common trigrams are, in decreasing order,

THE, ING, AND, HER, ERE, ENT, THA, NTH, WAS, ETH, FOR.

Armed with this information about English we are now able to examine and break a number of historical ciphers.

7.2. Shift Cipher

We first present one of the earliest ciphers, called the shift cipher. Encryption is performed by replacing each letter by the letter located a certain number of places further on in the alphabet. So for example if the key was three, then the plaintext A would be replaced by the ciphertext D, the letter B would be replaced by E and so on. The plaintext word HELLO would be encrypted as the ciphertext KHOOR. When this cipher is used with the key three, it is often called the Caesar cipher, although in many books the name Caesar cipher is sometimes given to the shift cipher with

Bigram	Freq. (%)	Bigram	Freq. (%)
TH	3.15	HE	2.51
AN	1.72	IN	1.69
ER	1.54	RE	1.48
ES	1.45	ON	1.45
EA	1.31	TI	1.28
AT	1.24	ST	1.21
EN	1.20	ND	1.18

TABLE 7.2. English bigram frequencies

any key. Strictly this is not correct since we only have evidence that Julius Caesar used the cipher with the key three.

There is a more mathematical explanation of the shift cipher which will be instructive for future discussions. First we need to identify each letter of the alphabet with a number. It is usual to identify the letter A with the number 0, the letter B with number 1, the letter C with the number 2 and so on until we identify the letter Z with the number 25. After we convert our plaintext message into a sequence of numbers, the ciphertext in the shift cipher is obtained by adding to each number the secret key k modulo 26, where the key is a number in the range 0 to 25. In this way we can interpret the shift cipher as a *stream cipher*, with keystream given by the repeating sequence

$$k, k, k, k, k, k, \ldots$$

This keystream is not very random, which results in it being easy to break the shift cipher. A naive way of breaking the shift cipher is to simply try each of the possible keys in turn, until the correct one is found. There are only 26 possible keys so the time for this exhaustive key search is very small, particularly if it is easy to recognize the underlying plaintext when it is decrypted.

We shall show how to break the shift cipher by using the statistics of the underlying language. Whilst this is not strictly necessary for breaking this cipher, later we shall see a cipher that is made up of a number of shift ciphers applied in turn and then the following statistical technique will be useful. Using a statistical technique on the shift cipher is also instructive as to how statistics of the underlying plaintext can arise in the resulting ciphertext. Take the following example ciphertext, which since it is public knowledge we represent in blue.

GB OR, BE ABG GB OR: GUNG VF GUR DHRFGVBA:
JURGURE 'GVF ABOYRE VA GUR ZVAQ GB FHSSRE
GUR FYVATF NAQ NEEBJF BS BHGENTRBHF SBEGHAR,
BE GB GNXR NEZF NTNVAFG N FRN BS GEBHOYRF,
NAQ OL BCCBFVAT RAQ GURZ? GB QVR: GB FYRRC;
AB ZBER; NAQ OL N FYRRC GB FNL JR RAQ
GUR URNEG-NPUR NAQ GUR GUBHFNAQ ANGHENY FUBPXF
GUNG SYRFU VF URVE GB, 'GVF N PBAFHZZNGVBA
QRIBHGYL GB OR JVFU'Q. GB QVR, GB FYRRC;
GB FYRRC: CREPUNAPR GB QERNZ: NL, GURER'F GUR EHO;
SBE VA GUNG FYRRC BS QRNGU JUNG QERNZF ZNL PBZR
JURA JR UNIR FUHSSYRQ BSS GUVF ZBEGNY PBVY,
ZHFG TVIR HF CNHFR: GURER'F GUR ERFCRPG
GUNG ZNXRF PNYNZVGL BS FB YBAT YVSR;

One technique used in breaking the previous sample ciphertext is to notice that the ciphertext still retains details about the word lengths of the underlying plaintext. For example the ciphertext

letter N appears as a single letter word. Since the only common single-letter words in English are A and I we can conclude that the key is either 13, since N is thirteen letters on from A in the alphabet, or 5, since N is five letters on from I in the alphabet. Hence, the moral here is to always remove word breaks from the underlying plaintext before encrypting using the shift cipher. But even if we ignore this information about the word break, we can still break this cipher using frequency analysis.

We compute the frequencies of the letters in the ciphertext and compare them with the frequencies obtained from English which we saw in Figure 7.1. We present the two bar graphs one above each other in Figure 7.2 so you can see that one graph looks almost like a shift of the other graph. The statistics obtained from the sample ciphertext are given in blue, whilst the statistics obtained from the underlying plaintext language are given in red. Note, we do not compute the red statistics from the actual plaintext since we do not know this yet, we only make use of the knowledge of the underlying language.

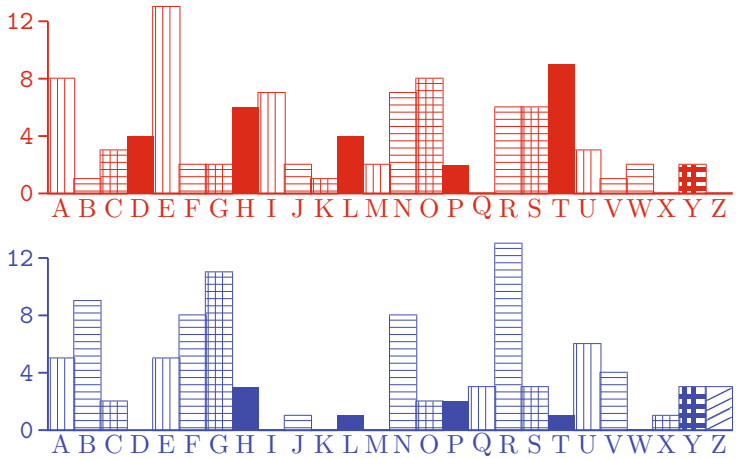


FIGURE 7.2. Comparison of plaintext and ciphertext frequencies for the shift cipher example

By comparing the two bar graphs in Figure 7.2 we can see by how much we think the blue graph has been shifted compared with the red graph. By examining where we think the plaintext letter E may have been shifted, one can hazard a guess that it is shifted by one of

$$2, 9, 13 \text{ or } 23.$$

Then by trying to deduce by how much the plaintext letter A has been shifted we can guess that it has been shifted by one of

$$1, 6, 13 \text{ or } 17.$$

The only shift value which is consistent appears to be the value 13, and we conclude that this is the most likely key value.

One may ask whether there is a more scientific way of performing the above comparison of bar graphs. Indeed there is, using something called the statistical distance. Let X and Y be random variables distributed according to distributions D_1 and D_2; we let V denote the support of X and Y (i.e. the set of values which can occur for X or Y with non-zero probability). We then define the statistical distance (actually the *total variation distance*, as there are many different statistical distances one can define) by

$$\Delta[X, Y] = \frac{1}{2} \sum_{u \in V} \left| \Pr_{X \leftarrow D_1}[X = u] - \Pr_{Y \leftarrow D_2}[Y = u] \right|.$$

To apply this to our example we let X denote the probabilities of letters occuring in English, i.e. the probabilities from Table 7.1, and we let Y_k denote the probabilities obtained from the ciphertext but shifted by the key value k. So we have twenty-six different distributions Y_k to compare to the fixed distribution X. The one which has the smallest distance is the one most likely to be the key.

Applying this method in this example we find the statistical distances given in Table 7.3. The value for the key 13 is significantly smaller than the values for the other keys; thus we can conclude (using this more scientific method) that the key is 13.

k	$\Delta(X, Y_k)$	k	$\Delta(X, Y_k)$
0	48.4	13	10.8
1	44.6	14	44.8
2	44.0	15	57.0
3	49.5	16	55.3
4	53.2	17	47.0
5	52.9	18	48.5
6	46.0	19	49.1
7	53.9	20	45.3
8	52.7	21	56.4
9	43.8	22	51.6
10	51.3	23	47.5
11	56.8	24	43.8
12	46.7	25	45.2

TABLE 7.3. Statistical distance between X and Y_k for the shift cipher example

We can now decrypt the ciphertext, using this key. This reveals that the underlying plaintext is the following text from Shakespeare's *Hamlet*:

> To be, or not to be: that is the question:
> Whether 'tis nobler in the mind to suffer
> The slings and arrows of outrageous fortune,
> Or to take arms against a sea of troubles,
> And by opposing end them? To die: to sleep;
> No more; and by a sleep to say we end
> The heart-ache and the thousand natural shocks
> That flesh is heir to, 'tis a consummation
> Devoutly to be wish'd. To die, to sleep;
> To sleep: perchance to dream: ay, there's the rub;
> For in that sleep of death what dreams may come
> When we have shuffled off this mortal coil,
> Must give us pause: there's the respect
> That makes calamity of so long life;

7.3. Substitution Cipher

The main problem with the shift cipher is that the number of keys is too small; we only have 26 possible keys. To increase the number of keys the *substitution cipher* was invented. To write down a key for the substitution cipher we first write down the alphabet, and then a permutation of the

alphabet directly below it. This mapping gives the substitution we make between the plaintext and the ciphertext

Plaintext alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet	GOYDSIPELUAVCRJWXZNHBQFTMK

Encryption involves replacing each letter in the top row by its value in the bottom row. Decryption involves first looking for the letter in the bottom row and then seeing which letter in the top row maps to it. Hence, the plaintext word HELLO would encrypt to the ciphertext ESVVJ if we used the substitution given above.

The number of possible keys is equal to the total number of permutations on 26 letters, namely the size of the group S_{26}, which is

$$26! \approx 4.03 \cdot 10^{26} \approx 2^{88}.$$

Since, as a rule of thumb, it is only feasible to run a computer on a problem which takes under 2^{80} steps we can deduce that this large key space is far too large to enable a brute force search even using a modern computer. Despite this we can break substitution ciphers using statistics of the underlying plaintext language, just as we did for the shift cipher.

Whilst the shift cipher can be considered as a stream cipher since the ciphertext is obtained from the plaintext by combining it with a keystream, the substitution cipher operates much more like a modern block cipher, with a block length of one English letter. A ciphertext block is obtained from a plaintext block by applying some (admittedly simple in this case) key-dependent algorithm.

Substitution ciphers are the ciphers commonly encountered in puzzle books; they have an interesting history and have occurred many times in literature. See for example the Sherlock Holmes story *The Adventure of the Dancing Men* by Arthur Conan Doyle; the plot of this story rests on a substitution cipher where the ciphertext characters are taken from an alphabet of "stick men" in various positions. The method of breaking the cipher as described by Holmes to Watson in this story is precisely the method we shall adopt below.

Example: We give a detailed example, which we make slightly easier by keeping in the ciphertext details about the underlying word spacing used in the plaintext. This is only for ease of exposition; the techniques we describe can still be used if we ignore these word spacings, although more care and thought is required. Consider the ciphertext

XSO MJIWXVL JODIVA STW VAO VY OZJVCO'W LTJDOWX KVAKOAXJTXIVAW VY SIDS XOKSAVLVDQ IAGZWXJQ. KVUCZXOJW, KVUUZAIKTXIVAW TAG UIKJVOLOKXJVAIKW TJO HOLL JOCJOWOAXOG, TLVADWIGO GIDIXTL UOGIT, KVUCZXOJ DTUOW TAG OLOKXJVAIK KVUUOJKO. TW HOLL TW SVWXIAD UTAQ JOWOTJKS TAG CJVGZKX GONOLVCUOAX KOAXJOW VY UTPVJ DLVMTL KVUCTAIOW, XSO JODIVA STW T JTCIGLQ DJVHIAD AZUMOJ VY IAAVNTXINO AOH KVUCTAIOW. XSO KVUCZXOJ WKIOAKO GOCTJXUOAX STW KLVWO JOLTXIVAWSICW HIXS UTAQ VY XSOWO VJDTAIWTXIVAW NIT KVLLTMVJTXINO CJVPOKXW, WXTYY WOKVAGUOAXW TAG NIWIXIAD IAGZWXJITL WXTYY. IX STW JOKOAXLQ IAXJVGZKOG WONOJTL UOKSTAIWUW YVJ GONOLVCIAD TAG WZCCVJXIAD OAXJOCJOAOZJITL WXZGOAXW TAG WXTYY, TAG TIUW XV CLTQ T WIDAIYIKTAX JVLO IA XSO GONOLVCUOAX VY SIDS-XOKSAVLVDQ IAGZWXJQ IA XSO JODIVA.

XSO GOCTJXUOAX STW T LTJDO CJVDJTUUO VY JOWOTJKS WZCCVJXOG MQ IAGZWXJQ, XSO OZJVCOTA ZAIVA, TAG ZE DVNOJAUOAX JOWOTJKS OWXTMLIWSUOAXW TAG CZMLIK KVJCVJTXIVAW. T EOQ OLOUOAX VY XSIW IW XSO WXJVAD LIAEW XSTX XSO GOCTJXUOAX STW HIXS XSO KVUCZXOJ, KVUUZAIKTXIVAW, UIKJVOLOKXJVAIKW TAG UOGIT IAGZWXJIOW IA XSO MJIWXVL JODIVA . XSO TKTGOUIK JOWOTJKS CJVDJTUUO IW VJDTAIWOG IAXV WONOA DJVZCW, LTADZTDOW

TAG TJKSIXOKXZJO, GIDIXTL UOGIT, UVMILO TAG HOTJTMLO KVUCZXIAD, UTKSIAO LOTJAIAD, RZTAXZU KVUCZXIAD, WQWXOU NOJIYIKTXIVA, TAG KJQCXVDJTCSQ TAG IAYVJUTXIVA WOKZJIXQ.

We can compute the following frequencies for single letters in the above ciphertext.

Letter	Freq. (%)	Letter	Freq. (%)	Letter	Freq. (%)
A	8.6995	B	0.0000	C	3.0493
D	3.1390	E	0.2690	F	0.0000
G	3.6771	H	0.6278	I	7.8923
J	7.0852	K	4.6636	L	3.5874
M	0.8968	N	1.0762	O	11.479
P	0.1793	Q	1.3452	R	0.0896
S	3.5874	T	8.0717	U	4.1255
V	7.2645	W	6.6367	X	8.0717
Y	1.6143	Z	2.7802		

In addition we determine that the most common bigrams in this piece of ciphertext are

TA, AX, IA, VA, WX, XS, AG, OA, JO, JV,

whilst the most common trigrams are

OAX, TAG, IVA, XSO, KVU, TXI, UOA, AXS.

Since the ciphertext letter O occurs with the greatest frequency, namely 11.479, we can guess that the ciphertext letter O corresponds to the plaintext letter E. We now look at what this means for two of the common trigrams found in the ciphertext

- The ciphertext trigram OAX corresponds to E * *.
- The ciphertext trigram XSO corresponds to * * E.

We examine similar common trigrams in English, which start or end with the letter E. We find that three common ones are given by ENT, ETH and THE. Since in the ciphertext trigrams we have one letter, X, in the first position in one and the last position in the other, we look for a similar letter in the English trigrams. We can conclude that it is highly likely that we have the correspondence

- X = T,
- S = H,
- A = N.

Even after this small piece of analysis we find that it is much easier to understand what the underlying plaintext should be. If we focus on the first two sentences of the ciphertext we are trying to break, and we change the letters for which we think we have found the correct mappings, then we obtain:

THE MJIWTVL JEDIVN HTW VNE VY EZJVCE'W LTJDEWT KVNKENTJTTIV NW VY HIDH TEKHNVLVDQ INGZWTJQ. KVUCZTEJW, KVUUZNIKTTIVNW TNG UIKJVELEKTJVNIKW TJE HELL JECJEWENTEG, TLVNDWIGE GIDITTL UEGIT, KVUCZTEJ DTUEW TNG ELEKTJVNIK KVUUEJKE.

Recall, this was after the four substitutions

O = E, X = T, S = H, A = N.

We now cheat and use the fact that we have retained the word sizes in the ciphertext. We see that since the letter T occurs as a single ciphertext letter we must have

T = I or T = A.

The ciphertext letter T occurs with a probability of 8.0717, which is the highest probability left, hence we are far more likely to have

$$\mathrm{T} = \mathrm{A}.$$

We have already considered the most popular trigram in the ciphertext so turning our attention to the next most popular trigram we see that it is equal to TAG which we suspect corresponds to the plaintext AN*. Therefore it is highly likely that G = D, since AND is a popular trigram in English. Our partially decrypted ciphertext is now equal to

THE MJIWTVL JEDIVN HAW VNE VY EZJVCE'W LAJDEWT KVNKENTJATIV NW VY HIDH TEKHNVLVDQ INDZWTJQ. KVUCZTEJW, KVUUZNIKATIVNW AND UIKJVELEKTJVNIKW AJE HELL JECJEWENTED, ALVNDWIDE DIDITAL UEDIA, KVUCZTEJ DAUEW AND ELEKTJVNIK KVUUEJKE.

This was after the six substitutions

$$\mathrm{O} = \mathrm{E},\ \mathrm{X} = \mathrm{T},\ \mathrm{S} = \mathrm{H},$$
$$\mathrm{A} = \mathrm{N},\ \mathrm{T} = \mathrm{A},\ \mathrm{G} = \mathrm{D}.$$

We now look at two-letter words which occur in the ciphertext:

- IX
  This corresponds to the plaintext *T. Therefore the ciphertext letter I must be one of the plaintext letters A or I, since the only common two-letter words in English ending in T are AT and IT. We already have worked out what the plaintext character A corresponds to, hence we must have I = I.
- XV
  This corresponds to the plaintext T*. Hence, we must have V = O.
- VY
  This corresponds to the plaintext O*. Hence, the ciphertext letter Y probably corresponds to one of F, N or R. We already know the ciphertext letter corresponding to N. In the ciphertext the probability of Y occurring is 1.6, but in English we expect F to occur with probability 2.2 and R to occur with probability 6.0. Hence, it is more likely that Y = F.
- IW
  This corresponds to the plaintext I*. Therefore, the plaintext character W must be one of F, N, S and T. We already have F, N, T, hence W = S.

All these deductions leave the partial ciphertext as

THE MJISTOL JEDION HAS ONE OF EZJOCE'S LAJDEST KONKENTJATIONS OF HIDH TEKHNOLODQ INDZSTJQ. KOUCZTEJS, KOUUZNIKATIONS AND UIKJOELEKTJONIKS AJE HELL JECJESENTED, ALONDSIDE DIDITAL UEDIA, KOUCZTEJ DAUES AND ELEKTJONIK KOUUEJKE.

This was after the ten substitutions

$$\mathrm{O} = \mathrm{E},\ \mathrm{X} = \mathrm{T},\ \mathrm{S} = \mathrm{H},\ \mathrm{A} = \mathrm{N},\ \mathrm{T} = \mathrm{A},$$
$$\mathrm{G} = \mathrm{D},\ \mathrm{I} = \mathrm{I},\ \mathrm{V} = \mathrm{O},\ \mathrm{Y} = \mathrm{F},\ \mathrm{W} = \mathrm{S}.$$

Even with half the ciphertext letters determined it is now quite easy to understand the underlying plaintext, taken from the website of the University of Bristol Computer Science Department circa 2001. We leave it to the reader to determine the final substitutions and recover the plaintext completely.

7.4. Vigenère Cipher

The problem with the shift cipher and the substitution cipher was that each plaintext letter always encrypted to the same ciphertext letter. Hence underlying statistics of the language could be used to break the cipher. For example it was easy to determine which ciphertext letter corresponded

to the plaintext letter E. From the early 1800s onwards, cipher designers tried to break this link between the plaintext and ciphertext.

The substitution cipher we used above was a mono-alphabetic substitution cipher, in that only one alphabet substitution was used to encrypt the whole alphabet. One way to solve our problem is to take a number of substitution alphabets and then encrypt each letter with a different alphabet. Such a system is called a polyalphabetic substitution cipher.

For example we could take

Plaintext alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet one	TMKGOYDSIPELUAVCRJWXZNHBQF
Ciphertext alphabet two	DCBAHGFEMLKJIZYXWVUTSRQPON

Then we encrypt the plaintext letters in odd-numbered positions encrypt using the first ciphertext alphabet, whilst we encrypt the plaintext letters in even-numbered positions using the second alphabet. For example, the plaintext word HELLO would encrypt to SHLJV, using the above two alphabets. Notice that the two occurrences of L in the plaintext encrypt to two different ciphertext characters. Thus we have made it harder to use the underlying statistics of the language. If one now does a naive frequency analysis one no longer obtains a common ciphertext letter corresponding to the plaintext letter E.

Essentially we are encrypting the message two letters at a time, hence we have a block cipher with block length two English characters. In real life one may wish to use around five rather than just two alphabets and the resulting key becomes very large indeed. With five alphabets the total key space is

$$(26!)^5 \approx 2^{441},$$

but the user only needs to remember the key which is a sequence of

$$26 \cdot 5 = 130$$

letters. However, just to make life hard for the attacker, the number of alphabets in use should also be hidden from his view and form part of the key. But for the average user in the early 1800s this was far too unwieldy a system, since the key was too hard to remember.

Despite its shortcomings the most famous cipher during the nineteenth century was based on precisely this principle. The *Vigenère cipher*, invented in 1533 by Giovan Battista Bellaso, was a variant on the above theme, but the key was easy to remember. When looked at in one way the Vigenère cipher is a polyalphabetic block cipher, but when looked at in another, it is a stream cipher; which is a natural generalization of the shift cipher.

The description of the Vigenère cipher as a block cipher takes the description of the polyalphabetic cipher above but restricts the possible ciphertext alphabets to one of the 26 possible cyclic shifts of the standard alphabet. Suppose five alphabets were used, this reduces the key space down to

$$26^5 \approx 2^{23}$$

and the size of the key to be remembered to a sequence of five numbers between 0 and 25.

However, the description of the Vigenère cipher as a stream cipher is much more natural. Just like the shift cipher, the Vigenère cipher again identifies letters with the numbers $0, \ldots, 25$. The secret key is a short sequence of letters (e.g. a word) which is repeated again and again to form a keystream. Encryption involves adding the plaintext letter to a key letter. Thus if the key is SESAME, encryption works as follows,

```
THISISATESTMESSAGE
SESAMESESAMESESAME
LLASUWSXWSFQWWKASI
```

Again we notice that A will encrypt to a different letter depending on where it appears in the message.

But the Vigenère cipher is still easy to break using the underlying statistics of English. Once we have found the length of the keyword, breaking the ciphertext is the same as breaking the shift cipher a number of times.

Example: As an example, suppose the ciphertext is given by

UTPDHUG NYH USVKCG MVCE FXL KQIB. WX RKU GI TZN, RLS BBHZLXMSNP KDKS; CEB IH HKEW IBA, YYM SBR PFR SBS, JV UPL O UVADGR HRRWXF. JV ZTVOOV YH ZCQU Y UKWGEB, PL UQFB P FOUKCG, TBF RQ VHCF R KPG, OU KFT ZCQU MAW QKKW ZGSY, FP PGM QKFTK UQFB DER EZRN, MCYE, MG UCTFSVA, WP KFT ZCQU MAW KQIJS. LCOV NTHDNV JPNUJVB IH GGV RWX ONKCGTHKFL XG VKD, ZJM VG CCI MVGD JPNUJ, RLS EWVKJT ASGUCS MVGD; DDK VG NYH PWUV CCHIIY RD DBQN RWTH PFRWBBI VTTK VCGNTGSF FL IAWU XJDUS, HFP VHCF, RR LAWEY QDFS RVMEES FZB CHH JRTT MVGZP UBZN FD ATIIYRTK WP KFT HIVJCI; TBF BLDPWPX RWTH ULAW TG VYCHX KQLJS US DCGCW OPPUPR, VG KFDNUJK GI JIKKC PL KGCJ IAOV KFTR GJFSAW KTZLZES WG RWXWT VWTL WP XPXGG, CJ FPOS VYC BTZCUW XG ZGJQ PMHTRAIBJG WMGFG. JZQ DPB JVYGM ZCLEWXR: CEB IAOV NYH JIKKC TGCWXF UHF JZK.

WX VCU LD YITKFTK WPKCGVCWIQT PWVY QEBFKKQ, QNH NZTTW IRFL IAS VFRPE ODJRXGSPTC EKWPTGEES, GMCG
TTVVPLTFFJ; YCW WV NYH TZYRWH LOKU MU AWO, KFPM VG BLTP VQN RD DSGG AWKWUKKPL KGCJ, XY OPP KPG ONZTT ICUJCHLSF KFT DBQNJTWUG. DYN MVCK ZT MFWCW HTWF FD JL, OPU YAE CH LQ! PGR UF, YH MWPP RXF CDJCGOSF, XMS UZGJQ JL, SXVPN HBG!

There is a way of finding the length of the keyword, which is repeated to form the keystream, called the *Kasiski test*. First we need to look for repeated sequences of characters. Recall that English has a large repetition of certain bigrams or trigrams and over a long enough string of text these are likely to match up to the same two or three letters in the key every so often. By examining the distance between two repeated sequences we can guess the length of the keyword. Each of these distances should be a multiple of the keyword, hence taking the greatest common divisor of all distances between the repeated sequences should give a good guess as to the keyword length.

Let us examine the above ciphertext and look for the bigram WX. The gaps between some of the occurrences of this bigram are 9, 21, 66 and 30, some of which may have occurred by chance, whilst some may reveal information about the length of the keyword. We now take the relevant greatest common divisors to find,

$$\gcd(30, 66) = 6, \text{ and } \gcd(3, 9) = \gcd(9, 66) = \gcd(9, 30) = \gcd(21, 66) = 3.$$

We are unlikely to have a keyword of length three so we conclude that the gaps of 9 and 21 occurred purely by chance. Hence, our best guess for the keyword is that it is of length six.

Now we take every sixth letter and look at the statistics just as we did for a shift cipher to deduce the first letter of the keyword. We can now see the advantage of using the histograms or statistical distance to break the shift cipher earlier. If we used the naive method and tried each of the 26 keys in turn we could still not detect which key is correct, since every sixth letter of an English sentence does not produce an English sentence. Thus we need to resort to using histograms or the statistical distance used earlier.

The relevant bar charts for every sixth letter starting with the first are given in Figure 7.3. We look for the possible locations of the three peaks corresponding to the plaintext letters A, E and T. We see that this sequence seems to be shifted by two positions in the blue graph compared with the red graph. Hence we can suspect that the first letter of the keyword is C, since C corresponds to a shift of two. Computing the statistical distance between the frequency of letters in English, and those of every sixth letter in the ciphertext shifted by a key k, produces the results in Table 7.4. Which indeed confirms our guess that the first letter of the keyword is C.

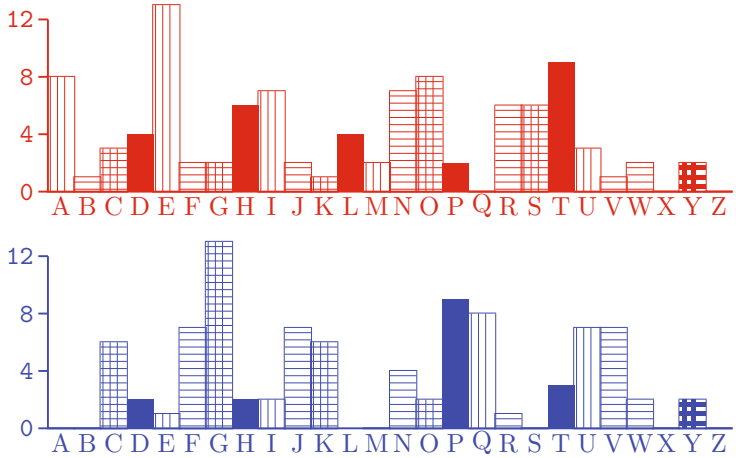


FIGURE 7.3. Comparison of plaintext and ciphertext frequencies for every sixth letter of the Vigenère example, starting with the first letter

k	$\Delta(X, Y_k)$	k	$\Delta(X, Y_k)$
0	60.7	13	40.9
1	42.8	14	47.2
2	12.4	15	50.4
3	45.0	16	46.8
4	59.5	17	41.5
5	52.6	18	45.7
6	48.0	19	55.8
7	47.8	20	54.0
8	50.5	21	46.2
9	47.1	22	47.8
10	54.7	23	48.4
11	53.5	24	43.2
12	47.8	25	53.8

TABLE 7.4. Statistical distance between X and Y_k for every sixth letter letter in the Vigenère example, starting with the first letter

We perform a similar step for every sixth letter, starting with the second one. The resulting bar graphs are given in Figure 7.4. Using the same technique we find that the blue graph appears to have been shifted along by 17 spaces, which corresponds to the second letter of the keyword being equal to R. Computing the statistical distance in Table 7.5 again confirms this guess.

Continuing in a similar way for the remaining four letters of the keyword we find the keyword is

CRYPTO.

The underlying plaintext is then found to be the following text from *A Christmas Carol* by Charles Dickens.

Scrooge was better than his word. He did it all, and infinitely more; and to Tiny Tim, who did not die, he was a second father. He became as good a friend, as good a master, and as good a man,

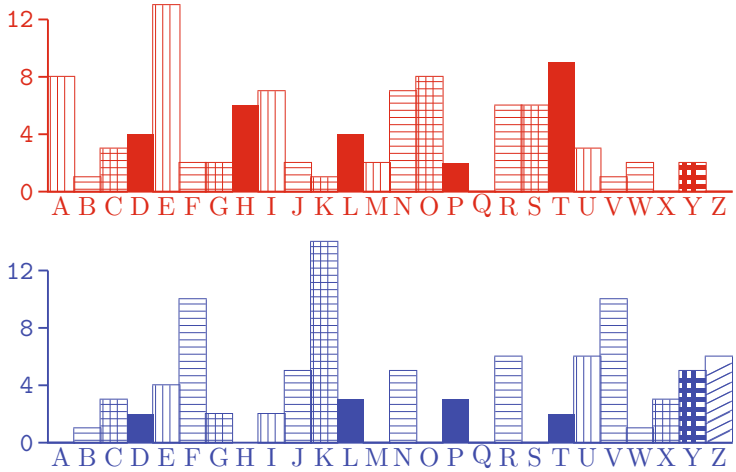


FIGURE 7.4. Comparison of plaintext and ciphertext frequencies for every sixth letter of the Vigenère example, starting with the second letter

k	$\Delta(X, Y_k)$	k	$\Delta(X, Y_k)$
0	54.6	13	39.5
1	46.2	14	55.1
2	38.3	15	59.3
3	45.1	16	53.2
4	52.6	17	17.8
5	48.3	18	47.1
6	34.6	19	53.4
7	45.3	20	53.6
8	52.4	21	44.0
9	51.3	22	49.4
10	44.6	23	48.6
11	53.2	24	48.1
12	47.1	25	52.2

TABLE 7.5. Statistical distance between X and Y_k for every sixth letter letter in the Vigenère example, starting with the second letter

as the good old city knew, or any other good old city, town, or borough, in the good old world. Some people laughed to see the alteration in him, but he let them laugh, and little heeded them; for he was wise enough to know that nothing ever happened on this globe, for good, at which some people did not have their fill of laughter in the outset; and knowing that such as these would be blind anyway, he thought it quite as well that they should wrinkle up their eyes in grins, as have the malady in less attractive forms. His own heart laughed: and that was quite enough for him.

He had no further intercourse with Spirits, but lived upon the Total Abstinence Principle, ever afterwards; and it was always said of him, that he knew how to keep Christmas well, if any man alive possessed the knowledge. May that be truly said of us, and all of us! And so, as Tiny Tim observed, God bless Us, Every One!

7.5. A Permutation Cipher

The ideas behind substitution-type ciphers forms part of the design of modern symmetric systems. For example, later we shall see that both DES and AES make use of a component called an S-Box, which is simply a substitution. The other component that is used in modern symmetric ciphers is based on permutations.

Permutation ciphers have been around for a number of centuries. Here we shall describe the simplest, which is particularly easy to break. We first fix a permutation group S_n for a small value of n, and a permutation $\sigma \in S_n$. It is the value of σ which will be the secret key. As an example suppose we take

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 3 & 5 \end{pmatrix} = (1243) \in S_5.$$

Now take some plaintext, say

Once upon a time there was a little girl called Snow White.

We break the text into chunks of five letters, and remove capitalisations,

onceu ponat imeth erewa salit tlegi rlcal ledsn owwhi te.

We first pad the message, with some random letters, so that we have a multiple of five letters in total

onceu ponat imeth erewa salit tlegi rlcal ledsn owwhi teahb.

Then we take each five-letter chunk in turn and swap the letters around according to our secret permutation σ. With our example permutation, we obtain

coenu npaot eitmh eewra lsiat etgli crall dlsdn wohwi atheb.

We then remove the spaces, so as to hide the value of n, producing the ciphertext

coenunpaoteitmheewralsiatetglicralldlsdnwohwiatheb.

However, breaking a permutation cipher is easy with a chosen plaintext attack, assuming the group of permutations used (i.e. the value of n) is reasonably small. To attack this cipher we mount a chosen plaintext attack, i.e. the attacker selects a plaintext of their choosing and asks for the encryption of it. In this specific example, they ask one of the parties to encrypt the message

abcdefghijklmnopqrstuvwxyz,

to obtain the ciphertext

cadbehfigjmknlorpsqtwuxvyz.

We can then deduce that the permutation looks something like

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \dots \\ 2 & 4 & 1 & 3 & 5 & 7 & 9 & 6 & 8 & 10 & 12 & 14 & 11 & 13 & 15 & \dots \end{pmatrix}.$$

We see that the sequence repeats (modulo 5) after every five steps and so the value of n is probably equal to five. We can recover the key by simply taking the first five columns of the above permutation.

Chapter Summary

- Many early ciphers can be broken because they do not successfully hide the underlying statistics of the language.

- Important principles behind early ciphers are those of substitution and permutation.
- Ciphers can either work on blocks of characters via some keyed algorithm or simply consist of adding some keystream to each plaintext character.

Further Reading

The best book on the history of ciphers is that by Kahn. It is a weighty tome, so those wishing a more rapid introduction should consult the book by Singh. The book by Churchhouse also gives an overview of a number of historical ciphers.

R. Churchhouse. *Codes and Ciphers. Julius Caesar, the Enigma and the Internet.* Cambridge University Press, 2001.

D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.* Scribner, 1996.

S. Singh. *The Codebook: The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography.* Doubleday, 2000.

CHAPTER 8

The Enigma Machine

Chapter Goals

- To explain the working of the Enigma machine.
- To explain how the German military used the Enigma machine during World War II, in particular how session keys were transmitted from the sender to the receiver.
- To explain how this enabled Polish and later British cryptanalysts to read the German traffic.
- To explain the use of the Bombe in mounting known plaintext attacks.

8.1. Introduction

With the advent of the 1920s people saw the need for a mechanical encryption device. Taking a substitution cipher and then rotating it was identified as an ideal solution. This idea had actually been used previously in a number of manual ciphers, but mechanization was able to make it far more efficient. The rotors could be implemented using wires and then encryption could be done mechanically using an electrical circuit.

By rotating the rotor we obtain a new substitution cipher. As an example, suppose the rotor used to produce the substitutions is given by the following values in the first position:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
TMKGOYDSIPELUAVCRJWXZNHBQF.

To encrypt the first letter we use the substitutions given above; i.e. we substitute B by M and Y by Q. However, to encrypt the second letter we rotate the rotor by one position, i.e. we move the bottom row one step to the left, and so use the substitutions

ABCDEFGHIJKLMNOPQRSTUVWXYZ
MKGOYDSIPELUAVCRJWXZNHBQFT,

whilst for the third letter we use the substitutions

ABCDEFGHIJKLMNOPQRSTUVWXYZ
KGOYDSIPELUAVCRJWXZNHBQFTM,

and so on. This gives us a polyalphabetic substitution cipher with 26 different alphabets.

The most famous of these machines was the Enigma machine used by Germany in World War II. We shall describe the most simple version of Enigma which only used three such rotors, chosen from the following set of five:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGDQVZNTOWYHXUSPAIBRCJ
AJDKSIRUXBLHWTMCQGZNPYFVOE
BDFHJLCPRTXVZNYEIWGAKMUSQO
ESOVPZJAYQUIRHXLNFTGKDCMWB
VZBRGITYUPSDNHLXAWMJQOFECK.

Machines in use towards the end of the war had a larger number of rotors, chosen from a larger set. Note that the order of the rotors in the machine is important, so the number of ways of choosing the rotors is

$$5 \cdot 4 \cdot 3 = 60.$$

Each rotor had an initial starting position, and since there are 26 possible starting positions for each rotor, the total number of possible starting positions is $26^3 = 17\,576$.

The first rotor would step on the second rotor on each full iteration under the control of a ring hitting a notch; likewise the stepping of the third rotor was controlled by the second rotor. Both the rings were movable and their positions again formed part of the key, although only the notch and ring positions for the first two rotors were important. Hence, the number of ring positions was $26^2 = 676$. The second rotor also had a "kick" associated with it, making the cycle length of the three rotors equal to

$$26 \cdot 25 \cdot 26 = 16\,900.$$

The effect of the moving rotors was that a given plaintext letter would encrypt to a different ciphertext letter on each press of the keyboard. Finally, a plugboard was used to swap letters twice in each encryption and decryption operation. This increased the complexity and gave another possible 10^{14} keys. The rotors used, their order, their starting positions, the ring positions and the plugboard settings all made up the secret key. Hence, the total number of keys was then around 2^{75}. To make sure encryption and decryption were the same operation a reflector was used. This was a fixed public substitution given by

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
YRUHQSLDPXNGOKMIEBFZCWVJAT.
```

To encrypt a plaintext character it would first be passed through the plugboard (thus possibly swapping it to another letter), then it passed forwards through the rotors, then through the reflector, then backwards through the rotors, and finally it would pass once more through the plugboard.

The operation of a simplified four-letter Enigma machine is depicted in Figure 8.1. By tracing the red lines one can see how the plaintext character A encrypts to the ciphertext character D. Notice that decryption can be performed with the machine in the same configuration as used for encyption. Now assume that rotor one moves on one step, so A now maps to D under rotor one, B to A, C to C and D to B. You should work out what happens with the example when we encrypt A again.

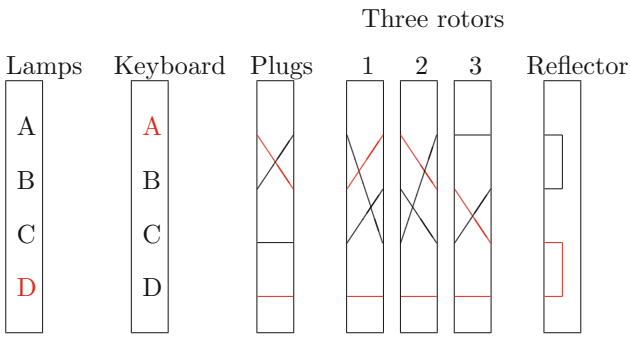


FIGURE 8.1. Simplified Enigma machine

In the rest of this chapter we present more details of the Enigma machine and some of the attacks which can be performed on it. However before presenting the machine itself we need to

fix some notation which will be used throughout this chapter. In particular lower-case letters will denote variables, upper-case letters will denote "letters" (of the plaintext/ciphertext languages) and Greek letters will denote permutations in S_{26} which we shall represent as permutations on the upper case letters. Hence x can equal X and Y, but X can only ever represent X, whereas χ could represent (XY) or (ABC).

Permutations will usually be given in cycle notation. One always has to make a choice as to whether we multiply permutations from left to right, or right to left. We decide to use the left-to-right method, hence

$$(ABCD)(BE)(CD) = (AEBD).$$

Permutations hence act on the right of letters, something we will denote by x^σ, e.g.

$$A^{(ABCD)(XY)} = B.$$

This is consistent with the usual notation of right action for groups acting on sets. See the appendix for more details about permutations.

We now collect some basic facts and theorems about permutations which we will need in the sequel.

Theorem 8.1. *Two permutations σ and τ which are conjugate, i.e. ones for which $\sigma = \lambda \cdot \tau \cdot \lambda^{-1}$ for some permutation λ, have the same cycle structure.*

We define the support of a permutation to be the set of letters which are not fixed by the permutation. Hence, if σ acts on the set of letters $\mathcal{L}$, then as usual we denote by $\mathcal{L}^\sigma$ the set of fixed points and hence the support is given by

$$\mathcal{L} \setminus \mathcal{L}^\sigma.$$

Theorem 8.2. *If two permutations, with the same support, consist only of disjoint transpositions then their product contains an even number of disjoint cycles of the same length. Conversely, if a permutation with support an even number of symbols has an even number of disjoint cycles of the same length, then the permutation can be written as a product of two permutations each of which consists of disjoint transpositions.*

Solving a Conjugation Problem: In many places we need an algorithm to solve the following problem: Given $\alpha_i, \beta_i \in S_{26}$, for $i = 1, \ldots, m$ find $\gamma \in S_{26}$ such that

$$\alpha_i = \gamma^{-1} \cdot \beta_i \cdot \gamma \text{ for } i = 1 \ldots, m.$$

Whilst there could be many such solutions γ, in the situations to which we will apply it we expect there to be only a few. For example, suppose we have one such equation with

$$\begin{aligned} \alpha_1 &= (AFCNE)(BWXHUJOG)(DVIQZ)(KLMYTRPS), \\ \beta_1 &= (AEYSXWUJ)(BFZNO)(CDPKQ)(GHIVLMRT) \end{aligned}$$

We need to determine the structure of the permutation γ such that

$$\alpha_1 = \gamma^{-1} \cdot \beta_1 \cdot \gamma.$$

We first look at what A should map to under γ. Suppose $A^\gamma = B$; then we have the equations

$$A^{\gamma \cdot \alpha_i} = B^{\alpha_i} = W \text{ and } A^{\beta_1 \cdot \gamma} = E^\gamma.$$

Thus we have $E^\gamma = W$. We then look at the equations

$$E^{\gamma \cdot \alpha_i} = W^{\alpha_i} = X \text{ and } E^{\beta_1 \cdot \gamma} = Y^\gamma.$$

So we have $Y^\gamma = X$. Continuing in this way via a pruned depth-first search we can determine a set of possible values for γ. Such an algorithm is relatively simple to write down in C, using a recursive procedure call. However, it is, of course, a bit of a pain to do this by hand, as would have been the only option in the 1930s and 1940s.

8.2. An Equation for the Enigma

To aid our discussion in later sections we now describe the Enigma machine as a permutation equation. We first assume a canonical map between letters and the integers $\{0, 1, \ldots, 25\}$ such that 0 is A, 1 is B etc. and we assume a standard three-wheel Enigma machine.

The wheel which turns the fastest we shall call rotor one, whilst the one which turns the slowest we shall call rotor three. This means that, when looking at a real machine rotor three is the leftmost rotor and rotor one is the rightmost rotor. Please keep this in mind as it can cause confusion (especially when reading day/message settings). The basic permutations which make up the Enigma machine are as follows.

Choice of Rotors: We assume that the three rotors are chosen from the following set of five rotors, presented in the table below. The Germans labelled these rotors I, II, III, IV and V, and they are the ones used in the actual Enigma machines. Each rotor also has a different notch position which controls how the stepping of one rotor drives the stepping of the others.

Rotor	Permutation Representation	Notch Position
I	$(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)$	16, i.e. Q
II	$(BJ)(CDKLHUP)(ESZ)(FIXVYOMW)(GR)(NT)$	4, i.e. E
III	$(ABDHPEJT)(CFLVMZOYQIRWUKXSG)$	21, i.e. V
IV	$(AEPLIYWCOXMRFZBSTGJQNH)(DV)(KU)$	9, i.e. J
V	$(AVOLDRWFIUQ)(BZKSMNHYC)(EGTJPX)$	25, i.e. Z

Reflector: A number of different reflectors were used in actual Enigma machines. In our description we shall use the reflector given earlier, which is often referred to as "Reflector B". This reflector has representation via disjoint cycles as

$$\varrho = (AY)(BR)(CU)(DH)(EQ)(FS)(GL)(IP)(JX)(KN)(MO)(TZ)(VW).$$

An Enigma Key: An Enigma key consists of the following information:

- A choice of rotors ρ_1, ρ_2, ρ_3 from the above choice of five possible rotors. Note that this choice of rotors affects the three notch positions, which we shall denote by n_1, n_2 and n_3. Also, as noted above, the rotor ρ_3 is placed in the left of the actual machine, whilst rotor ρ_1 is placed on the right. Hence, if in a German code book it says use rotors

  $$I, II, III,$$

  this means in our notation that ρ_1 is selected to be rotor III, that ρ_2 is selected to be rotor II and ρ_3 is selected to be rotor I.
- One must also select the ring positions, which we shall denote by r_1, r_2 and r_3. In the actual machine these are letters, but we shall use our canonical numbering to represent these as integers in $\{0, 1, \ldots, 25\}$.
- The plugboard is simply a product of disjoint transpositions which we shall denote by the permutation τ. In what follows we shall denote a plug linking letter A with letter B by $A \leftrightarrow B$.
- The starting rotor positions we shall denote by p_1, p_2 and p_3. These are the letters which can be seen through the windows on the top of the Enigma machine. Remember our numbering system is that the window on the left corresponds to p_3 whilst the one on the right corresponds to p_1.

The Encryption Operation: We let σ denote the shift-up permutation given by

$$\sigma = (ABCDEFGHIJKLMNOPQRSTUVWXYZ).$$

The stepping of the second and third rotor is probably the hardest part to grasp when first looking at an Enigma machine, however this has a relatively simple description when one looks at it in a mathematical manner.

Given the above description of the key we wish to deduce the permutation ϵ_j, which represents, for $j = 0, 1, 2, \ldots$, the encryption of the jth letter. We first set

$$\begin{aligned} m_1 &= n_1 - p_1 - 1 \pmod{26}, \\ m &= n_2 - p_2 - 1 \pmod{26}, \\ m_2 &= m_1 + 1 + 26 \cdot m. \end{aligned}$$

The values of m_1 and m_2 control the stepping of the second and third rotors.

We let $\lfloor x \rfloor$ denote the round towards zero function, i.e. $\lfloor 1.9 \rfloor = 1$ and $\lfloor -1.9 \rfloor = -1$. We now set, for encrypting letter j,

$$\begin{aligned} k_1 &= \lfloor (j - m_1 + 26)/26 \rfloor, \\ k_2 &= \lfloor (j - m_2 + 650)/650 \rfloor, \\ i_1 &= p_1 - r_1 + 1, \\ i_2 &= p_2 - r_2 + k_1 + k_2, \\ i_3 &= p_3 - r_3 + k_2. \end{aligned}$$

Notice how i_3 is stepped on every $650 = 26 \cdot 25$ iterations whilst i_2 is stepped on every 26 iterations and also stepped on an extra notch every 650 iterations.

We can now present ϵ_j as

$$\begin{aligned} \epsilon_j = \; & \tau \cdot (\sigma^{i_1+j} \cdot \rho_1 \cdot \sigma^{-i_1-j}) \cdot (\sigma^{i_2} \cdot \rho_2 \cdot \sigma^{-i_2}) \cdot (\sigma^{i_3} \cdot \rho_3 \cdot \sigma^{-i_3}) \cdot \varrho \\ & \cdot (\sigma^{i_3} \cdot {\rho_3}^{-1} \cdot \sigma^{-i_3}) \cdot (\sigma^{i_2} \cdot {\rho_2}^{-1} \cdot \sigma^{-i_2}) \cdot (\sigma^{i_1+j} \cdot {\rho_1}^{-1} \cdot \sigma^{-i_1-j}) \cdot \tau. \end{aligned}$$

Note that the same equation/machine is used to encrypt the jth letter as is used to decrypt the jth letter. Hence we have

$${\epsilon_j}^{-1} = \epsilon_j.$$

Also note that each ϵ_j consists of a product of disjoint transpositions. We shall always use γ_j to represent the internal rotor part of the Enigma machine, hence

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau.$$

8.3. Determining the Plugboard Given the Rotor Settings

For the moment, assume that we know values for the rotor order, ring settings and rotor positions; for this purpose we are given γ_j. We would like to determine the plugboard settings. The goal is therefore to determine τ given some information about ϵ_j for some values of j. One often sees it written that determining the plugboard given the rotor settings is equivalent to solving a substitution cipher. This is true, but the methods given in some sources are too simplistic.

Let m denote the actual message being encrypted, c the corresponding ciphertext, and m' the ciphertext decrypted under the cipher with no plugboard, i.e. with the obvious notation,

$$\begin{aligned} m &= c^{\epsilon}, \\ m' &= c^{\gamma}. \end{aligned}$$

The following is an example value of m' for a plugboard containing only one plug

ZNCT UPZN A EIME, THEKE WAS A GILL CALLED SNZW WHFTE.

I have left the spacings in the English words. You may then deduce that Z should really map to O, or T should really map to E, or E should really map to T, or maybe K should really map to R, etc. But which should be the correct plug setting to achieve this mapping? The actual correct plug setting is that O should map to Z; the other mappings are the result of this single plug setting. We now present some ways of obtaining information about the plugboard given various scenarios.

8.3.1. Ciphertext Only Attack: In a ciphertext only attack one can proceed as one would for a normal substitution cipher. We need a method to distinguish something which could be natural language from something which is completely random. The best option seems to be to use something called the Sinkov statistic. Let f_i, for $i = A, \ldots, Z$, denote the frequencies of the various letters in standard English. For a given piece of text we let n_i, for $i = A, \ldots, Z$, denote the frequencies of the various letters within the sample piece of text. The Sinkov statistic for the sample text is given by

$$s = \sum_{i=A}^{Z} n_i \cdot f_i.$$

The higher the value of s, the more likely that the text represents an extract from a natural language.

To mount a ciphertext only attack we let γ_j denote our current approximation for ϵ_j (initially γ_j has no plug settings, but this will change as the method progresses). We now go through all possible single-plug settings, $\alpha^{(k)}$. There are $26 \cdot 25/2 = 325$ of these. We then decrypt the ciphertext c using the cipher

$$\alpha^{(k)} \cdot \gamma_j \cdot \alpha^{(k)}.$$

This results in 325 possible plaintext messages $m^{(k)}$ for $k = 1, \ldots, 325$. For each one of these we compute the Sinkov statistic s_k, and keep the value of $\alpha^{(k)}$ which results in s_k being maximized. We then set our new γ_j to be $\alpha^{(k)} \cdot \gamma_j \cdot \alpha^{(k)}$ and repeat until no further improvement can be made in the test statistic.

This methodology seems very good at finding the missing plugboard settings. Suppose we are given that the day setting is

Rotors	Rings	Pos	Plugboard
III, II, I	PPD	MDL	Unknown

The actual hidden plugboard is given by $A \leftrightarrow B$, $C \leftrightarrow D$, $E \leftrightarrow F$, $G \leftrightarrow H$, $I \leftrightarrow J$ and $K \leftrightarrow L$. We obtain the ciphertext

```
HUCDODANDHOMYXUMGLREDSQQJDNJAEXUKAZOYGBYLEWFNWIBWILSMAETFFBVPR
GBYUDNAAIEVZZKCUFNIUTOKNKAWUTUWQJYAUHMFWJNIQHAYNAGTDGTCTNYKTCU
FGYQBSRRUWZKZFWKPGVLUHYWZCZSOYJNXHOSKVPHGSGSXEOQWOZYBXQMKQDDXM
BJUPSQODJNIYEPUCEXFRHDQDAQDTFKPSZEMASWGKVOXUCEYWBKFQCYZBOGSFES
OELKDUTDEUQZKMUIZOGVTWKUVBHLVXMIXKQGUMMQHDLKFTKRXCUNUPPFKWUFCU
PTDMJBMMTPIZIXINRUIEMKDYQFMIQAEVLWJRCYJCUKUFYPSLQUEZFBAGSJHVOB
CHAKHGHZAVJZWOLWLBKNTHVDEBULROARWOQGZLRIQBVVSNKRNUCIKSZUCXEYBD
QKCVMGLRGFTBGHUPDUHXIHLQKLEMIZKHDEPTDCIPF
```

The plugboard settings are found in the following order $I \leftrightarrow J$, $E \leftrightarrow F$, $A \leftrightarrow B$, $G \leftrightarrow H$, $K \leftrightarrow L$ and $C \leftrightarrow D$. The plaintext is determined to be:

```
ITWASTHEBESTOFTIMESITWASTHEWORSTOFTIMESITWASTHEAGEOFWISDOMITWA
STHEAGEOFFOOLISHNESSITWASTHEEPOCHOFBELIEFITWASTHEEPOCHOFINCRED
ULITYITWASTHESEASONOFLIGHTITWASTHESEASONOFDARKNESSITWASTHESPRI
NGOFHOPEITWASTHEWINTEROFDESPAIRWEHADEVERYTHINGBEFOREUSWEHADNOT
HINGBEFOREUSWEWEREALLGOINGDIRECTTOHEAVENWEWEREALLGOINGDIRECTTH
EOTHERWAYINSHORTTHEPERIODWASSOFARLIKETHEPRESENTPERIODTHATSOMEO
```

FITSNOISIESTAUTHORITIESINSISTEDONITSBEINGRECEIVEDFORGOODORFOREVILINTHESUPERLATIVEDEGREEOFCOMPARISONONLY

8.3.2. Known Plaintext Attack: When one knows the plaintext there is a choice of two methods one can employ. The first method is simply based on a depth-first search technique, whilst the second makes use of some properties of the encryption operation.

Technique One: In the first technique we take each wrong letter in turn, from our current approximation γ_j to ϵ_j. In the above example of the encryption of the first sentences from "A Tale of Two Cities", we have that the first ciphertext letter H should map to the plaintext letter I. This implies that the plugboard must contain plug settings $H \leftrightarrow p_H$ and $I \leftrightarrow p_I$, for letters p_H and p_I with

$$p_H{}^{\gamma_0} = p_I.$$

In a similar manner we deduce the following further equations:

$$\begin{aligned} p_U{}^{\gamma_1} &= p_T, \quad p_C{}^{\gamma_2} = p_W, \quad p_D{}^{\gamma_3} = p_A, \\ p_O{}^{\gamma_4} &= p_S, \quad p_D{}^{\gamma_5} = p_T, \quad p_A{}^{\gamma_6} = p_H, \\ p_N{}^{\gamma_7} &= p_E, \quad p_D{}^{\gamma_8} = p_B, \quad p_H{}^{\gamma_9} = p_E. \end{aligned}$$

The various permutations representing the first few γ_js for the given rotor and ring positions are as follows:

$$\begin{aligned} \gamma_0 &= (AW)(BH)(CZ)(DE)(FT)(GJ)(IN)(KL)(MQ)(OV)(PU)(RS)(XY), \\ \gamma_1 &= (AZ)(BL)(CE)(DH)(FK)(GJ)(IS)(MX)(NQ)(OY)(PR)(TU)(VW), \\ \gamma_2 &= (AZ)(BJ)(CV)(DW)(EP)(FX)(GO)(HS)(IY)(KL)(MN)(QT)(RU), \\ \gamma_3 &= (AF)(BC)(DY)(EO)(GU)(HK)(IV)(JR)(LX)(MN)(PW)(QS)(TZ), \\ \gamma_4 &= (AJ)(BD)(CF)(EL)(GN)(HX)(IM)(KQ)(OS)(PV)(RT)(UY)(WZ), \\ \gamma_5 &= (AW)(BZ)(CT)(DI)(EH)(FV)(GU)(JO)(KP)(LN)(MX)(QY)(RS), \\ \gamma_6 &= (AL)(BG)(CO)(DV)(EN)(FS)(HY)(IZ)(JT)(KW)(MP)(QR)(UX), \\ \gamma_7 &= (AI)(BL)(CT)(DE)(FN)(GH)(JY)(KZ)(MO)(PS)(QX)(RU)(VW), \\ \gamma_8 &= (AC)(BH)(DU)(EM)(FQ)(GV)(IO)(JZ)(KS)(LT)(NR)(PX)(WY), \\ \gamma_9 &= (AB)(CM)(DY)(EZ)(FG)(HN)(IR)(JX)(KV)(LW)(OT)(PQ)(SU). \end{aligned}$$

We now proceed as follows: suppose we know that exactly six plugs are being used. This means that if we pick a letter at random, say T, then there is a $14/26 = 0.53$ chance that this letter is not plugged to another one. Let us therefore make this assumption for the letter T, in which case $p_T = T$. From the above equations involving γ_1 and γ_5 we then deduce that

$$p_U = U \text{ and } p_D = C.$$

We then use the equations involving γ_3 and γ_8, since we now know p_D, to deduce that

$$p_A = B \text{ and } p_B = A.$$

These last two checks are consistent, so we can assume that our original choice of $p_T = T$ was a good one. From the equations involving γ_6, using $p_A = B$ we deduce that

$$p_H = G.$$

Using this in the equations involving γ_0 and γ_9 we deduce that

$$p_I = J \text{ and } p_E = F.$$

We then find that our five plug settings of $A \leftrightarrow B$, $C \leftrightarrow D$, $E \leftrightarrow F$, $G \leftrightarrow H$ and $I \leftrightarrow J$ allow us to decrypt the first ten letters correctly. To deduce the final plug setting will require a longer piece of ciphertext, and the corresponding piece of known plaintext.

This technique can also be used when one knows partial information about the rotor positions. For example, many of the following techniques will allow us to deduce the differences $p_i - r_i$, but not the actual values of r_i or p_i. However, by applying the above technique, on assuming $r_i =$'A', we will at some point deduce a contradiction. At this point we know that either a rotor turnover has occurred incorrectly, or one has not occurred when it should have done. Hence, we can at this point backtrack and deduce the correct turnover. For an example of this technique at work see the later section on the Bombe.

Technique Two: A second method is possible when fewer than thirteen plugs are used. In the plaintext obtained under γ_j a number of incorrect letters will appear. Again we let m denote the actual plaintext and m' the plaintext derived under the current (possibly empty) plugboard setting. We suppose that there are t plugs left to find.

Suppose we concentrate on each place for which the incorrect plaintext letter A occurs, i.e. all occurrences of A in the plaintext m which are wrong in m'. Let x denote the corresponding ciphertext letter. There are two possible cases which can occur

- The letter x should be plugged to an unknown letter. In this case the resulting letter in the message m' will behave randomly (assuming γ_j acts like a random permutation).
- The letter x does not occur in a plugboard setting. In this case the resulting incorrect plaintext character is the one which should be plugged to A in the actual cipher.

Assuming ciphertext letters are uniformly distributed, the first case will occur with probability $t/13$, whilst the alternative will occur with probability $1 - t/13$. This gives the following method for determining the letter to which A should be connected. For all occurances of A in the plaintext m compute the frequency of the corresponding letter in the approximate plaintext m'. The one with the highest frequency is highly likely to be the one which should be connected to A on the plugboard. Indeed we expect the proportion of such positions with the correct letter to be given by $1 - t/13$, whilst all other letters we expect to occur in proportions of $t/(13 \cdot 26)$ each.

The one problem with this second technique is that it requires a relatively large amount of known plaintext. Hence, in practice the first technique is more likely to be used.

Knowledge of ϵ_j for Some js: If we know the value of the permutation ϵ_j for values of $j \in \mathcal{S}$, then we have the following equation

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau \text{ for } j \in \mathcal{S}.$$

Since $\tau = \tau^{-1}$ we can thus compute possible values of τ using our previous method for solving this conjugation problem. This might not determine the whole plugboard but it will determine enough for other methods to be used.

Knowledge of $\epsilon_j \cdot \epsilon_{j+3}$ for Some js: A similar method to the previous one applies in this case as well, since, if we know $\epsilon_j \cdot \epsilon_{j+3}$ for all $j \in \mathcal{S}$ and we know γ_j, we can use the equation

$$(\epsilon_j \cdot \epsilon_{j+3}) = \tau \cdot (\gamma_j \cdot \gamma_{j+3}) \cdot \tau \text{ for } j \in \mathcal{S}.$$

The utility of a method upon knowing $\epsilon_j \cdot \epsilon_{j+3}$ will become apparent in a little while.

8.4. Double Encryption of Message Keys

The Polish mathematicians Jerzy Różycki, Henryk Zygalski and Marian Rejewski were the first to find ways of analysing the Enigma machine. To understand their methods one must first understand how the Germans used the machine. On each day the machine was set up with a key, as above, which was chosen by looking in a code book; each subnet would have a different day key.

To encipher a message the sending operator decided on a message key. The message key would be a sequence of three letters, say DHI, which would need to be transported to the recipient. Using the day key, the message key would be enciphered twice. The double enciphering was to act as a

form of error control. Hence, DHI might be enciphered as $XHJKLM$. Note that D encrypts first to X and then to K.

The receiver would obtain $XHJKLM$ and then decrypt this to obtain DHI. Both operators would then move the wheels around to the positions D, H and I, i.e. they would turn the wheels so that D was in the leftmost window, H in the middle one and I in the rightmost window. Then the actual message would be enciphered. For this example, in our notation, this would mean that the message key was equal to the day key, except that $p_1 = 8$, i.e. I, $p_2 = 7$, i.e. H and $p_3 = 3$, i.e. D.

Suppose we intercept a set of messages which have the following headers, consisting of the encryption of the three-letter rotor positions, followed by its encryption again, i.e. the first six letters of each message are equal to

UCWBLR	ZSETEY	SLVMQH	SGIMVW	PMRWGV
VNGCTP	OQDPNS	CBRVPV	KSCJEA	GSTGEU
DQLSNL	HXYYHF	GETGSU	EEKLSJ	OSQPEB
WISIIT	TXFEHX	ZAMTAM	VEMCSM	LQPFNI
LOIFMW	JXHUHZ	PYXWFQ	FAYQAF	QJPOUI
EPILWW	DOGSMP	ADSDRT	XLJXQK	BKEAKY
.....	.....	.....	.....	.....
DDESRY	QJCOUA	JEZUSN	MUXROQ	SLPMQI
RRONYG	ZMOTGG	XUOXOG	HIUYIE	KCPJLI
DSESEY	OSPPEI	QCPOLI	HUXYOQ	NYIKFW

Let us take the last one of these and look at it in more detail. We know that there are three underlying secret letters, say l_1, l_2 and l_3. We also know that

$$l_1{}^{\epsilon_0} = N,\ l_2{}^{\epsilon_1} = Y,\ l_3{}^{\epsilon_2} = I,$$

and

$$l_1{}^{\epsilon_3} = K,\ l_2{}^{\epsilon_4} = F,\ l_3{}^{\epsilon_5} = W.$$

Hence, given that ${\epsilon_j}^{-1} = \epsilon_j$, we have

$$N^{\epsilon_0\epsilon_3} = l_1{}^{\epsilon_0\epsilon_0\epsilon_3} = l_1{}^{\epsilon_3} = K,\ Y^{\epsilon_1\epsilon_4} = F,\ I^{\epsilon_2\epsilon_5} = W.$$

Continuing in this way we can compute a permutation representation of the three products as follows:

$$\begin{aligned}
\epsilon_0 \cdot \epsilon_3 &= (ADSMRNKJUB)(CV)(ELFQOPWIZT)(HY),\\
\epsilon_1 \cdot \epsilon_4 &= (BPWJUOMGV)(CLQNTDRYF)(ES)(HX),\\
\epsilon_2 \cdot \epsilon_5 &= (AC)(BDSTUEYFXQ)(GPIWRVHZNO)(JK).
\end{aligned}$$

8.5. Determining the Internal Rotor Wirings

However, life was even more difficult for the Polish analysts as they did not even know the rotor wirings or the reflector values. Hence, they needed to break the scheme without even having a description of the actual machine. They did at least have access to a non-military version of Enigma and deduced the basic structure. In this they had two bits of luck:

(1) They deduced that the wiring between the plugboard and the rightmost rotor was in alphabetical order. Had this not been the case they would have needed to find an additional, hidden permutation.
(2) Secondly, the French cryptographer Gustave Bertrand obtained from a German spy, Hans-Thilo Schmidt, two months' worth of day keys. Thus, for two months of traffic the Poles had access to the day settings.

From this information they needed to deduce the internal wirings of the Enigma machine.

Note that in the pre-war days the Germans only used three wheels out of a choice of three, hence the number of day keys is actually reduced by a factor of ten. This is, however, only a slight simplification (at least with modern technology). To explain the method suppose we are given that the day setting is

Rotors	Rings	Pos	Plugboard
III, II, I	TXC	EAZ	$(AMTEBC)$

We do not know what the actual rotors are at present, but we know that the one labelled rotor I will be placed in the rightmost slot (our label one). So we have

$$r_1 = 2,\ r_2 = 23,\ r_3 = 19, p_1 = 25,\ p_2 = 0,\ p_3 = 4.$$

Suppose also that the data from the previous section was obtained as traffic for that day. Hence, we obtain the following three values for the products $\epsilon_j \cdot \epsilon_{j+1}$,

$$\begin{aligned} \epsilon_0 \cdot \epsilon_3 &= (ADSMRNKJUB)(CV)(ELFQOPWIZT)(HY), \\ \epsilon_1 \cdot \epsilon_4 &= (BPWJUOMGV)(CLQNTDRYF)(ES)(HX), \\ \epsilon_2 \cdot \epsilon_5 &= (AC)(BDSTUEYFXQ)(GPIWRVHZNO)(JK). \end{aligned}$$

From these we wish to deduce the values of $\epsilon_0, \epsilon_1, \ldots, \epsilon_5$. We will use the fact that ϵ_j is a product of disjoint transpositions and Theorem 8.2.

We take the first product and look at it in more detail. We take the sets of two cycles of equal degree and write them above one another, with the bottom one reversed in order, i.e.

$$\begin{array}{cccccccccc} A & D & S & M & R & N & K & J & U & B \\ T & Z & I & W & P & O & Q & F & L & E \end{array} \qquad \begin{array}{cc} C & V \\ Y & H \end{array}$$

We now run through all possible shifts of the bottom rows. Each shift gives us a possible value of ϵ_0 and ϵ_3. The value of ϵ_0 is obtained from reading off the disjoint transpositions from the columns, the value of ϵ_3 is obtained by reading off the transpositions from the "off diagonals". For example with the above orientation we would have

$$\begin{aligned} \epsilon_0 &= (AT)(DZ)(SI)(MW)(RP)(NO)(KQ)(JF)(UL)(BE)(CY)(VH), \\ \epsilon_3 &= (DT)(SZ)(MI)(RW)(NP)(KO)(JQ)(UF)(BL)(AE)(VY)(CH). \end{aligned}$$

This still leaves us, in this case, with $20 = 2 \cdot 10$ possible values for ϵ_0 and ϵ_3.

Now, to reduce this number we need to rely on operational errors by German operators. Various operators had a tendency to always select the same three letter message key. For example, popular choices where QWE (the first letters on the keyboard). One operator used the letters of his girlfriend's name, Cillie, hence such "cribs" (or guessed/known plaintexts in today's jargon) became known as "cillies". Note, for our analysis here we only need one cillie for the day when we wish to obtain the internal wiring of rotor I.

In our dummy example, suppose we guess (correctly) that the first message key is indeed QWE, and that $UCWBLR$ is the encryption of QWE twice. This in turn tells us how to align our cycle of length 10 in the first permutation, as under ϵ_0 the letter Q must encrypt to U.

$$\begin{array}{cccccccccc} A & D & S & M & R & N & K & J & U & B \\ L & E & T & Z & I & W & P & O & Q & F \end{array}$$

We can check that this is consistent as we see that Q under ϵ_3 must then encrypt to B. Assuming we carry on in this way we will finally deduce that

$$\begin{aligned}
\epsilon_0 &= (AL)(BF)(CH)(DE)(GX)(IR)(JO)(KP)(MZ)(NW)(QU)(ST)(VY),\\
\epsilon_1 &= (AK)(BQ)(CW)(DM)(EH)(FJ)(GT)(IZ)(LP)(NV)(OR)(SX)(UY),\\
\epsilon_2 &= (AJ)(BN)(CK)(DZ)(EW)(FP)(GX)(HS)(IY)(LM)(OQ)(RU)(TV),\\
\epsilon_3 &= (AF)(BQ)(CY)(DL)(ES)(GX)(HV)(IN)(JP)(KW)(MT)(OU)(RZ),\\
\epsilon_4 &= (AK)(BN)(CJ)(DG)(EX)(FU)(HS)(IZ)(LW)(MR)(OY)(PQ)(TV),\\
\epsilon_5 &= (AK)(BO)(CJ)(DN)(ER)(FI)(GQ)(HT)(LM)(PX)(SZ)(UV)(WY).
\end{aligned}$$

We now need to use this information to deduce the value of ρ_1, etc. So for the rest of this section we assume that we know the ϵ_j for $j = 0, \ldots, 5$, and so we mark them in blue. Recall that we have

$$\begin{aligned}
\epsilon_j \;=\; & \tau \cdot (\sigma^{i_1+j} \cdot \rho_1 \cdot \sigma^{-i_1-j}) \cdot (\sigma^{i_2} \cdot \rho_2 \cdot \sigma^{-i_2}) \cdot (\sigma^{i_3} \cdot \rho_3 \cdot \sigma^{-i_3}) \cdot \varrho \\
& \cdot (\sigma^{i_3} \cdot {\rho_3}^{-1} \cdot \sigma^{-i_3}) \cdot (\sigma^{i_2} \cdot {\rho_2}^{-1} \cdot \sigma^{-i_2}) \cdot (\sigma^{i_1+j} \cdot {\rho_1}^{-1} \cdot \sigma^{-i_1-j}) \cdot \tau
\end{aligned}$$

We now assume that no stepping of the second rotor occurs during the first six encryptions under the day setting. This holds with quite high probability, namely $20/26 \approx 0.77$. Should the assumption turn out to be false we will notice in our later analysis and it will mean that we can deduce something about the (unknown to us at this point) position of the notch on the first rotor.

Given that we know the day settings, including τ and the values of i_1, i_2 and i_3 (since we are assuming $k_1 = k_2 = 0$ for $0 \le j \le 5$), we can write the above equation for $0 \le j \le 5$ as

$$\begin{aligned}
\lambda_j &= \sigma^{-i_1-j} \cdot \tau \cdot \epsilon_j \cdot \tau \cdot \sigma^{i_1+j} \\
&= \rho_1 \cdot \sigma^{-j} \cdot \gamma \cdot \sigma^{j} \cdot {\rho_1}^{-1}.
\end{aligned}$$

Where λ_j is now known and we wish to determine ρ_1 for some fixed but unknown value of γ. The permutation γ is in fact equal to

$$\gamma = (\sigma^{i_2-i_1} \cdot \rho_2 \cdot \sigma^{-i_2}) \cdot (\sigma^{i_3} \cdot \rho_3 \cdot \sigma^{-i_3}) \cdot \varrho \cdot (\sigma^{i_3} \cdot {\rho_3}^{-1} \cdot \sigma^{-i_3}) \cdot (\sigma^{i_2} \cdot {\rho_2}^{-1} \cdot \sigma^{i_1-i_2}).$$

In our example we get the following values for λ_j:

$$\begin{aligned}
\lambda_0 &= (AD)(BR)(CQ)(EV)(FZ)(GP)(HM)(IN)(JK)(LU)(OS)(TW)(XY),\\
\lambda_1 &= (AV)(BP)(CZ)(DF)(EI)(GS)(HY)(JL)(KO)(MU)(NQ)(RW)(TX),\\
\lambda_2 &= (AL)(BK)(CN)(DZ)(EV)(FP)(GX)(HS)(IY)(JM)(OQ)(RU)(TW),\\
\lambda_3 &= (AS)(BF)(CZ)(DR)(EM)(GN)(HY)(IW)(JO)(KQ)(LX)(PV)(TU),\\
\lambda_4 &= (AQ)(BK)(CT)(DL)(EP)(FI)(GX)(HW)(JU)(MO)(NY)(RS)(VZ),\\
\lambda_5 &= (AS)(BZ)(CV)(DO)(EM)(FR)(GQ)(HK)(IL)(JT)(NP)(UW)(XY).
\end{aligned}$$

We now form, for $j = 0, \ldots, 4$,

$$\begin{aligned}
\mu_j &= \lambda_j \cdot \lambda_{j+1}, \\
&= \rho_1 \cdot \sigma^{-j} \cdot \gamma \cdot \sigma^{-1} \cdot \gamma \cdot \sigma^{j+1} \cdot {\rho_1}^{-1}, \\
&= \rho_1 \cdot \sigma^{-j} \cdot \delta \cdot \sigma^{j} \cdot {\rho_1}^{-1},
\end{aligned}$$

where $\delta = \gamma \cdot \sigma^{-1} \cdot \gamma \cdot \sigma$ is unknown. Eliminating δ via $\delta = \sigma^{j-1} \cdot {\rho_1}^{-1} \cdot \mu_{j-1} \rho_1 \cdot \sigma^{-j+1}$ we find the following equations for $j = 1, \ldots, 4$,

$$\begin{aligned}
\mu_j &= (\rho_1 \cdot \sigma^{-1} \cdot {\rho_1}^{-1}) \cdot \mu_{j-1} \cdot (\rho_1 \cdot \sigma \cdot {\rho_1}^{-1}), \\
&= \alpha \cdot \mu_{j-1} \cdot \alpha^{-1},
\end{aligned}$$

where $\alpha = \rho_1 \cdot \sigma^{-1} \cdot {\rho_1}^{-1}$. Hence, μ_j and μ_{j-1} are conjugate and so by Theorem 8.1 have the same cycle structure. For our example we have

$$\begin{aligned}
\mu_0 &= (AFCNE)(BWXHUJOG)(DVIQZ)(KLMYTRPS),\\
\mu_1 &= (AEYSXWUJ)(BFZNO)(CDPKQ)(GHIVLMRT),\\
\mu_2 &= (AXNZRTIH)(BQJEP)(CGLSYWUD)(FVMOK),\\
\mu_3 &= (ARLGYWFK)(BIHNXDSQ)(CVEOU)(JMPZT),\\
\mu_4 &= (AGYPMDIR)(BHUTV)(CJWKZ)(ENXQSFLO).
\end{aligned}$$

At this point we can check whether our assumption of no-stepping, i.e. a constant value for the values of i_2 and i_3, is valid. If a step did occur in the second rotor then the above permutations would be unlikely to have the same cycle structure.

We need to determine the structure of the permutation α; this is done by looking at the four equations simultaneously. We note that σ and α are conjugates, under ρ_1, and we know that α has cycle structure of a single cycle of length 26, since σ is the shift-left permutation. In our example we only find one possible solution for α, namely

$$\alpha = (AGYWUJOQNIRLSXHTMKCEBZVPFD).$$

To solve for ρ_1 we need to find a permutation such that

$$\alpha = \rho_1 \cdot \sigma^{-1} \cdot {\rho_1}^{-1}.$$

We find there are 26 such solutions

$$
\begin{array}{l}
(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)\\
(AFHRVJ)(BLU)(CNXSTQYDGEMPIW)(KOZ)\\
(AGFIXTRWDHSUCO)(BMQZLVKPJ)(ENY)\\
(AHTSVLWEOBNZMRXUDIYFJCPKQ)\\
(AIZN)(BOCQ)(DJ)(EPLXVMSWFKRYGHU)\\
(AJEQCRZODKSXWGI)(BPMTUFLYHVN)\\
(AKTVOER)(BQDLZPNCSYI)(FMUGJ)(HW)\\
(AL)(BR)(CTWI)(DMVPOFN)(ESZQ)(GKUHXYJ)\\
(AMWJHYKVQFOGLBS)(CUIDNETXZR)\\
(ANFPQGMX)(BTYLCVRDOHZS)(EUJI)(KW)\\
(AOIFQH)(BUKX)(CWLDPREVS)(GN)(MY)(TZ)\\
(APSDQIGOJKYNHBVT)(CX)(EWMZUL)(FR)\\
(AQJLFSEXDRGPTBWNIHCYOKZVUM)\\
(ARHDSFTCZWOLGQK)(BXEYPUNJM)\\
(ASGRIJNKBYQLHEZXFUOMC)(DT)(PVW)\\
(ATE)(BZYRJONLIKC)(DUPWQM)(FVXGSH)\\
(AUQNMEB)(DVYSILJPXHGTFWRK)\\
(AVZ)(CDWSJQOPYTGURLKE)(FXIM)\\
(AWTHINOQPZBCEDXJRMGV)(FYUSK)\\
(AXKGWUTIORNP)(BDYV)(CFZ)(HJSLM)\\
(AYWVCGXLNQROSMIPBEF)(DZ)(HK)(JT)\\
(AZEGYXMJUVD)(BF)(CHLOTKIQSNRP)\\
(BGZFCIRQTLPD)(EHMKJV)(NSOUWX)\\
(ABHNTMLQUXOVFDCJWYZG)(EISP)\\
(ACKLRSQVGBITNUY)(EJXPF)(HOWZ)\\
(ADEKMNVHPGCLSRTOXQW)(BJY)(IUZ)
\end{array}
$$

These are the values of $\rho_1 \cdot \sigma^i$, for $i = 0, \ldots, 25$. So with one day's messages we can determine the value of ρ_1 up to multiplication by a power of σ. The Polish had access to two months' such data and so were able to determine similar sets for ρ_2 and ρ_3 (as different rotor orders are used on different days). Note that, at this point, the Germans did not use a selection of three from five rotors.

If we select three representatives $\hat{\rho_1}$, $\hat{\rho_2}$ and $\hat{\rho_3}$, from the sets of possible rotors, then we have

$$
\begin{aligned}
\hat{\rho_1} &= \rho_1 \cdot \sigma^{l_1},\\
\hat{\rho_2} &= \rho_2 \cdot \sigma^{l_2},\\
\hat{\rho_3} &= \rho_3 \cdot \sigma^{l_3}.
\end{aligned}
$$

However, we still do not know the value for the reflector ϱ, or the correct values of l_1, l_2 and l_3. To understand how to proceed we present the following theorem.

Theorem 8.3. *Consider an Enigma machine $\mathcal{E}$ that uses rotors ρ_1, ρ_2 and ρ_3, and reflector ϱ. Then there is an Enigma machine $\hat{\mathcal{E}}$ using rotors $\hat{\rho_1}$, $\hat{\rho_2}$ and $\hat{\rho_3}$, and a different reflector $\hat{\varrho}$ such that, for every setting of $\mathcal{E}$, there is a setting of $\hat{\mathcal{E}}$ such that the machines have identical behaviour.*

Furthermore, $\hat{\mathcal{E}}$ can be constructed so that the machines use identical day settings except for the ring positions.

PROOF. The following proof was shown to me by Eugene Luks; I thank him for allowing me to reproduce it here. The first claim is that $\hat{\varrho}$ is determined via

$$\hat{\varrho} = \sigma^{-(l_1+l_2+l_3)} \cdot \varrho \cdot \sigma^{-(l_1+l_2+l_3)}.$$

We can see this by the following argument (and the fact that the reflector is uniquely determined by the above equation). Define the following function

$$\begin{aligned} P(\phi_1, \phi_2, \phi_3, \psi, t_1, t_2, t_3) = \tau \cdot (\sigma^{t_1} \cdot \phi_1 \cdot \sigma^{-t_1}) \cdot (\sigma^{t_2} \cdot \phi_2 \cdot \sigma^{-t_2}) \cdot (\sigma^{t_3} \cdot \phi_3 \cdot \sigma^{-t_3}) \cdot \psi \\ \cdot (\sigma^{t_3} \cdot \phi_3^{-1} \cdot \sigma^{-t_3}) \cdot (\sigma^{t_2} \cdot \phi_2^{-1} \cdot \sigma^{-t_2}) \cdot (\sigma^{t_1} \cdot \phi_1^{-1} \cdot \sigma^{-t_1}) \cdot \tau \end{aligned}$$

We then have the relation,

$$P(\hat{\rho}_1, \hat{\rho}_2, \hat{\rho}_3, \hat{\varrho}, t_1, t_2, t_3) = P(\rho_1, \rho_2, \rho_3, \varrho, t_1, t_2 + l_1, t_3 + l_1 + l_2).$$

Recall the following expressions for the functions which control the stepping of the three rotors:

$$\begin{aligned} k_1 &= \lfloor (j - m_1 + 26)/26 \rfloor, \\ k_2 &= \lfloor (j - m_2 + 650)/650 \rfloor, \\ i_1 &= p_1 - r_1 + 1, \\ i_2 &= p_2 - r_2 + k_1 + k_2, \\ i_3 &= p_3 - r_3 + k_2. \end{aligned}$$

The Enigma machine $\mathcal{E}$ is given by the equation

$$\epsilon_j = P(\rho_1, \rho_2, \rho_3, \varrho, i_1 + j, i_2, i_3)$$

where we interpret i_2 and i_3 as functions of j as above. We now set the ring positions in $\hat{\mathcal{E}}$ to be given by

$$r_1, r_2 + l_1, r_3 + l_1 + l_2$$

in which case we have that the output of this Enigma machine is given by

$$\hat{\epsilon}_j = P(\hat{\rho}_1, \hat{\rho}_2, \hat{\rho}_3, \hat{\varrho}, i_1 + j, i_2 - l_1, i_3 - l_1 - l_2).$$

But then we conclude that $\epsilon_j = \hat{\epsilon}_j$. □

We now use this result to fully determine $\mathcal{E}$ from the available data. We pick values of $\hat{\rho}_1$, $\hat{\rho}_2$ and $\hat{\rho}_3$ and determine a possible reflector by solving for $\hat{\varrho}$ in

$$\begin{aligned} \epsilon_0 = \tau \cdot (\sigma^{i_1} \cdot \hat{\rho}_1 \cdot \sigma^{-i_1}) \cdot (\sigma^{i_2} \cdot \hat{\rho}_2 \cdot \sigma^{-i_2}) \cdot (\sigma^{i_3} \cdot \hat{\rho}_3 \cdot \sigma^{-i_3}) \cdot \hat{\varrho} \\ \cdot (\sigma^{i_3} \cdot \hat{\rho}_3{}^{-1} \cdot \sigma^{-i_3}) \cdot (\sigma^{i_2} \cdot \hat{\rho}_2{}^{-1} \cdot \sigma^{-i_2}) \cdot (\sigma^{i_1} \cdot \hat{\rho}_1{}^{-1} \cdot \sigma^{-i_1}) \cdot \tau \end{aligned}$$

We let $\hat{\mathcal{E}}^1$ denote the Enigma machine with rotors given by $\hat{\rho}_1, \hat{\rho}_2, \hat{\rho}_3$ and reflector $\hat{\varrho}$, but with ring settings the same as in the target machine $\mathcal{E}$ (we know the ring settings of $\mathcal{E}$ since we have the day key). Note that $\hat{\mathcal{E}}^1 \neq \hat{\mathcal{E}}$ from the above proof, since the rings are in the same place as in the target machine.

Assume we have obtained a long message, with a given message key. We put the machine $\hat{\mathcal{E}}^1$ in the message key configuration and start to decrypt the message. This will work (i.e. produce a valid decryption) up to the point when the sequence of permutations $\hat{\epsilon}^1_j$ produced by $\hat{\mathcal{E}}^1$ differs from the sequence ϵ_j produced by $\mathcal{E}$.

At this point we cycle through all values of l_1 and fix the first permutation (and also the associated reflector) to obtain a new Enigma machine $\hat{\mathcal{E}}^2$ which allows us to decrypt more of the long message. If a long enough message is obtained we can also obtain l_2 in this way, or alternatively

wait for another day when the rotor order is changed. Thus the entire internal workings of the Enigma machine can be determined.

8.6. Determining the Day Settings

Having now determined the internal wirings, given the set of two months of day settings obtained by Bertrand, the next task is to determine the actual key when the day settings are not available. At this stage we assume that the German operators are still using the "encrypt the message setting twice" routine. The essential trick here is to notice that if we write the cipher as

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau,$$

then

$$\epsilon_j \cdot \epsilon_{j+3} = \tau \cdot \gamma_j \cdot \gamma_{j+3} \cdot \tau.$$

So $\epsilon_j \cdot \epsilon_{j+3}$ is conjugate to $\gamma_j \cdot \gamma_{j+3}$ and so by Theorem 8.1 they have the same cycle structure. More importantly the cycle structure does not depend on the plugboard τ.

Hence, if we can use the cycle structure to determine the rotor settings then all that remains is to determine the plugboard settings. From the rotor settings we know the values of γ_j, for $j = 1, \ldots, 6$; from the encrypted message keys we can compute ϵ_j for $j = 1, \ldots, 6$ as in the previous section. Hence, the plugboard settings can be recovered by solving another of our conjugacy problems, for τ. This is easier than before as we have that τ must be a product of disjoint transpositions.

We have already discussed how to compute $\epsilon_j \cdot \epsilon_{j+3}$ from the encryption of the message keys. Hence, we *simply* compute these values and compare their cycle structures with those obtained by running through all possible

$$60 \cdot 26^3 \cdot 26^3 = 18\,534\,946\,560$$

choices for the rotors, positions and ring settings! Note that when this was done by the Polish analysts in the 1930s there was only a choice of the ordering of three rotors. The extra choice of rotors did not come in till a bit later. Hence, the total choice was 10 times less than this figure.

The above simplifies further if we assume that no stepping of the second and third rotor occurs during the calculation of the first six ciphertext characters. Recall this happens around seventy seven percent of the time. In such a situation the cycle structure depends only on the rotor order and the difference $p_i - r_i$ between the starting rotor position and the ring setting. Hence, we might as well assume that $r_1 = r_2 = r_3 = 0$ when computing all of the cycle structures. So, for seventy seven percent of days our search amongst the cycle structures is then only among

$$60 \cdot 26^3 = 1\,054\,560 \text{ (resp. } 105\,456)$$

possible cycle structures.

After the above procedure we have determined all values of the initial day setting bar p_i and r_i, however we know the differences $p_i - r_i$. We also know for any given message the message key p'_1, p'_2, p'_3. Hence, in breaking the actual message we only require the solution for r_1, r_2; the value for r_3 is irrelevant as the third rotor never moves a fourth rotor. Most German messages started with the same two-letter word followed by a space (space was encoded by 'X'). Hence, we only need to go through 26^2 different positions to get the correct ring setting. Actually one goes through 26^2 wheel positions with a fixed ring, and uses the differences to infer the true ring settings.

Once r_i is determined from one message the value of p_i can be determined for the day key and then all messages can be trivially broken. Another variant here, if a suitable piece of known plaintext can be deduced, is to apply the technique from Section 8.3.2 with the obvious modification to deduce the ring settings as well.

8.7. The Germans Make It Harder

In September 1938 the German operators altered the way that day and message keys were used. Now a day key consisted of a rotor order, the ring settings and the plugboard. But the rotor positions were not part of the day key. A cipher operator would now choose their own initial rotor positions, say AXE and their own message rotor positions, say GPI. The operator would put their machine in the AXE setting and then encrypt GPI twice as before, to obtain say $POWKNP$. The rotors would then be placed in the GPI position and the message would be encrypted. The message header would be $AXEPOWKNP$.

This procedure makes the analysis of the previous section useless, as each message would now have its own "day" rotor position setting, and so one could not collect data from many messages so as to recover $\epsilon_0 \cdot \epsilon_3$ etc. as in the previous section.

What was needed was a new way of characterizing the rotor positions. The strategy invented by Zygalski was to use so-called "females". In the six letters of the enciphered message key a female is the occurrence of the same letter in the same position in each substring of three. For example, the header $POWKNP$ contains no females, but the header $POWPNL$ contains one female in position zero, i.e. the repeated values of P, separated by three positions.

To consider what is implied by the existence of such females, firstly suppose we receive $POWPNL$ as above and that the unknown first key setting is x. Then we have that, if ϵ_i represents the Enigma machine in the day setting,

$$x^{\epsilon_0} = x^{\epsilon_3} = P,$$

that is

$$P^{\epsilon_0 \cdot \epsilon_3} = x^{\epsilon_0 \cdot \epsilon_0 \cdot \epsilon_3} = x^{\epsilon_3} = P.$$

In other words P is a fixed point of the permutation $\epsilon_0 \cdot \epsilon_3$.

Since the number of fixed points is a feature of the cycle structure and the cycle structure is invariant under conjugation, we see that the number of fixed points of $\epsilon_0 \cdot \epsilon_3$ is the same irrespective of the plugboard setting.

The use of such females was made easier by so-called Zygalski sheets. The following precomputation was performed, for each rotor order. An Enigma machine was set up with rings in position AAA and then, for each position A to Z of the third (leftmost) rotor a sheet was created. This sheet was a table of 51 by 51 squares, consisting of the letters of the alphabet repeated twice in each direction minus one row and column. A square was removed if the Enigma machine with first and second rotor with that row/column position had a fixed point in the permutation $\epsilon_0 \cdot \epsilon_3$. So for each rotor order there was a set of 26 sheets.

Note, we are going to use the sheets to compute the day ring setting, but the computation is done using different rotor positions but with a fixed ring setting. This is because it is easier with an Enigma machine to rotate the rotor positions than to change the ring settings. Then converting between ring and rotor settings is simple.

In fact, it makes sense to also produce a set of sheets for the permutation $\epsilon_1 \cdot \epsilon_4$ and $\epsilon_2 \cdot \epsilon_5$, as without these the number of keys found by the following method is quite large. Hence, for each rotor order we will have 26×3 perforated sheets. We now describe the method used by the Polish analysts when only three rotors were used (extending it to five rotors is simple but was time consuming at the time). We proceed via an example. Suppose a set of message headers are received in one day. From these we keep all those which possess a female in the part corresponding to the encryption of the message key. For example, we obtain the following message headers:

HUXTBPGNP	DYRHFLGFS	XTMRSZRCX	YGZVQWZQH
BILJWWRRW	QYRZXOZJV	SZYJPFBPY	MWIBUMWRM
YXMHCUHHR	FUGWINCIA	BNAXGHFGG	TLCXYUXYC
RELCOYXOF	XNEDLLDHK	MWCQOPQVN	AMQCZQCTR
MIPVRYVCR	MQYVVPVKA	TQNJSSIQS	KHMCKKCIL

LQUXIBFIV	NXRZNYXNV	AMUIXVVFV	UROVRUAWU
DSJVDFVTT	HOMFCSQCM	ZSCTTETBH	SJECXKCFN
UPWMQJMSA	CQJEHOVBO	VELVUOVDC	TXGHFDJFZ
DKQKFEJVE	SHBOGIOQQ	QWMUKBUVG	

Now assuming a given rotor order, say the rightmost rotor is rotor I, the middle one is rotor II and the leftmost rotor is rotor III, we remove all those headers which could have had a stepping action of the middle rotor in the first six encryptions. To compute these we take the third character of the above message headers, i.e. the position p_1 of the rightmost rotor in the encryption of the message key, and the position of the notch on the rightmost rotor assuming the rightmost rotor is I, i.e. $n_1 = 16$, i.e. Q. We compute the value of m_1 according to Section 8.2

$$m_1 = n_1 - p_1 - 1 \pmod{26}.$$

and remove all those for which

$$\lfloor (j - m_1 + 26)/26 \rfloor \neq 0 \text{ for } j = 0, 1, 2, 3, 4, 5.$$

This leaves us with the following message headers

HUXTBPGNP	DYRHFLGFS	YGZVQWZQH	QYRZXOZJV
SZYJPFBPY	MWIBUMWRM	FUGWINCIA	BNAXGHFGG
TLCXYUXYC	XNEDLLDHK	MWCQOPQVN	AMQCZQCTR
MQYVVPVKA	LQUXIBFIV	NXRZNYXNV	AMUIXVVFV
DSJVDFVTT	ZSCTTETBH	SJECXKCFN	UPWMQJMSA
CQJEHOVBO	TXGHFDJFZ	DKQKFEJVE	SHBOGIOQQ

We now consider each of the three sets of females in turn. For ease of discussion we only consider those corresponding to $\epsilon_0 \cdot \epsilon_3$. We therefore only examine those message headers which have the same letter in the fourth and seventh positions, i.e.

QYRZXOZJV	TLCXYUXYC	XNEDLLDHK	MWCQOPQVN
AMQCZQCTR	MQYVVPVKA	DSJVDFVTT	ZSCTTETBH
SJECXKCFN	UPWMQJMSA	SHBOGIOQQ	

We now perform the following operation, for each letter P_3. We take the Zygalski sheet for rotor order III, II, I, permutation $\epsilon_0 \cdot \epsilon_3$ and letter P_3 and we place this down on the table. We think of this first sheet as corresponding to the ring setting

$$r_3 = Q - Q = A,$$

where the Q comes from the first letter in the first message header, `QYRZXOZJV`. Each row r and column c of the first sheet corresponds to the ring setting

$$\begin{aligned} r_1 &= R - r, \\ r_2 &= Y - c. \end{aligned}$$

We now repeat the following process for each message header with a first letter which we have not met before. We take the first letter of the next message header, `TLCXYUXYC`, in this case T, and we take the sheet with label

$$P_3 + T - Q.$$

This sheet then has to be placed on top of the other sheets at a certain offset to the original sheet. The offset is computed by taking the top leftmost square of the new sheet and placing it on top of the square (r, c) of the first sheet where

$$\begin{aligned} r &= R - C, \\ c &= Y - L, \end{aligned}$$

i.e. we take the difference between the third (resp. second) letter of the new message header and the third (resp. second) letter of the first message header.

This process is repeated until all of the given message headers are used up. Any square which is now clear on all sheets then gives a possible setting for the rings for that day. The actual setting can be read off the first sheet using the correspondence above.

This process will give a relatively large number of possible ring settings for each possible rotor order. However, when we intersect the possible values obtained from considering the females in the 0/3 position with those in the 1/4 and the 2/5 positions we find that the number of possibilities shrinks dramatically. Often this allows us to uniquely determine the rotor order and ring setting for the day. We determine in our example that the rotor order is given by III, II and I, with ring settings given by $r_1 = A$, $r_2 = B$ and $r_3 = C$.

To determine the plugboard settings for the day we can either use a piece of known plaintext as before, or, if no such text is available, we can use the females to help drastically reduce the number of possibilities for the plugboard settings.

8.8. Known Plaintext Attack and the Bombes

Turing (among others) wanted a technique to break Enigma which did not rely on the way the German military used the system, which could and did change. Turing settled on a known plaintext attack, using what was known at the time as a "crib". A crib was a piece of plaintext which was suspected to lie in the given piece of ciphertext.

The methodology of this technique was, from a given piece of ciphertext and a suspected piece of corresponding plaintext, to first deduce a so-called "menu". A menu is simply a graph which represents the various relationships between ciphertext and plaintext letters. Then the menu was used to program an electro-mechanical device, called a Bombe. A Bombe was a device which enumerated the Enigma wheel positions and, given the data in the menu, deduced the possible settings for the rotor orders, wheel positions and some of the plugboard. Finally, the ring positions and the remaining parts of the plugboard needed to be found.

In the following we present a version of this technique which we have deduced from various sources. We follow a running example through so as to explain the method in more detail.

From Ciphertext to a Menu: Suppose we receive the following ciphertext

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
```

and that we know, for example because we suspect it to be a shipping forecast, that the ciphertext encrypts at some point the plaintext[1]

```
DOGGERFISHERGERMANBIGHTEAST
```

Now, we know that in the Enigma machine, a letter cannot decrypt to itself. This means that there are only a few positions for which the plaintext will align correctly with the ciphertext. Consider the following alignment:

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
-DOGGERFISHERGERMANBIGHTEAST---------------
```

then we see that this is impossible since the S in the plaintext FISHER cannot correspond to the S in the ciphertext. Continuing in this way we find that there are only six possible alignments of the plaintext fragment with the ciphertext:

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTEAST---------------
---DOGGERFISHERGERMANBIGHTEAST-------------
-----DOGGERFISHERGERMANBIGHTEAST-----------
--------DOGGERFISHERGERMANBIGHTEAST--------
----------DOGGERFISHERGERMANBIGHTEAST------
```

[1]This plaintext refers to sea regions in the BBC shipping weather forecast.

```
----------------DOGGERFISHERGERMANBIGHTEAST
```

In the following we will focus on the first alignment, i.e. we will assume that the first ciphertext letter H decrypts to D and so on. In practice the correct alignment out of all the possible ones would need to be deduced by skill, judgement and experience. However, in any given day a number of such cribs would be obtained and so only the most likely ones would be accepted for use in the following procedure.

As is usual with all our techniques there is a problem if the middle rotor turns over in the part of the ciphertext which we are considering. Our piece of chosen plaintext is 27 letters long, so we could treat it in two sections of 13 letters (and drop the last letter). The advantage of this is that we know the middle rotor will only advance once every 26 turns of the first rotor. Hence, by selecting two groups of roughly 13 letters we can obtain two possible alignments, one of which we know does not contain a middle rotor movement.

We therefore concentrate on the following two alignments:

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERG-----------------------------
-------------ERMANBIGHTEAS-----------------
```

We now deal with each alignment in turn and examine the various pairs of letters. We note that if H encrypts to D in the first position then D will encrypt to H in the same Enigma configuration. We make a record of the letters and the positions for which one letter encrypts to the other. These are placed in a graph with vertices corresponding to the letters and edges being labelled by the positions of the related encryptions. This results in the two graphs (or menus) given in Figures 8.2 and 8.3:

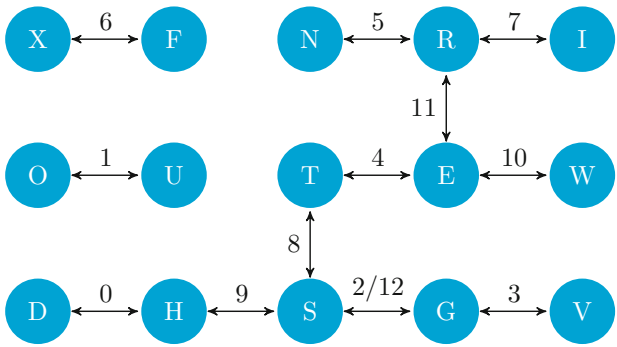


FIGURE 8.2. Menu 1

These menus tell us a lot about the configuration of the Enigma machine, in terms of its underlying permutations. Each menu is then used to program a Bombe. In fact we program one Bombe not only for each menu, but also for each possible rotor order. Thus if five rotor orders are in use, we need to program $2 \cdot 60 = 120$ such Bombes.

8.8.1. The Turing/Welchman Bombe: There are many descriptions of the Bombe as an electrical circuit. In the following we present the basic workings of the Bombe in terms of a modern computer; note however that in practice this is not very efficient. The Bombe's electrical circuit was able to execute the basic operations at the speed of light (i.e. the time it takes for a current to pass around a circuit), hence simulating this with a modern computer is very inefficient. The Bombe was in fact an electro-mechanical computer which was designed to perform a specific task – namely determining the wheel settings of the Enigma machine given one of the menus described

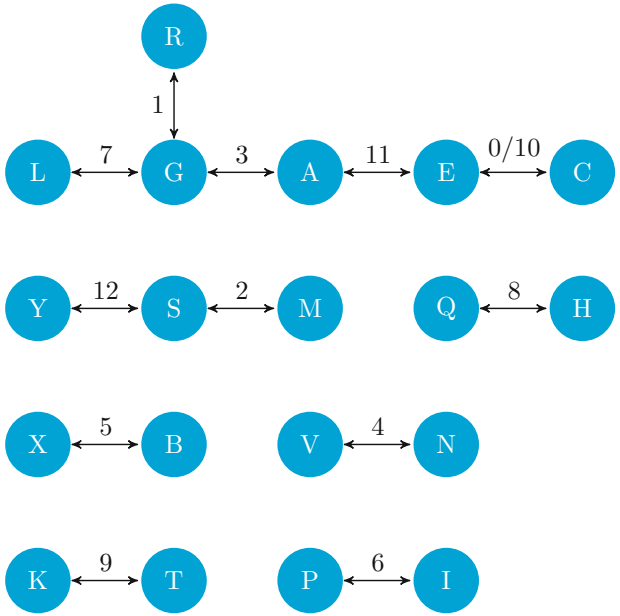


FIGURE 8.3. Menu 2

above. The initial design was made by Turing, but Welchman contributed a vital component (called the diagonal board) which made the process very efficient.

I have found that the best way to think of the Bombe is as a computer with 26 registers, each of which is 26 bits in length. In a single "step" of the Bombe a single bit in one register bank is set. Say we set bit F of register H; this corresponds to us wishing to test whether F is plugged to H in the actual Enigma configuration. This testing is done with the wheels in a given specific location. The Bombe then passes through a series of states until it stabilizes. In the actual Bombe this occurs at the speed of light; in a modern computer simulation it needs to be actually programmed and so occurs at the speed of a computer. Once the register bank stabilizes, each bit that is set means that if the tested condition is true, and the wheels are in the correct position, then so must this condition be true, i.e. if bit J of register K is set then J should be plugged to K in the Enigma machine. If we obtain a contradiction, then either the initial tested condition is false, or the wheels are in the wrong position. Continuing in this way with different wheel settings we either deduce the correct wheel settings, or deduce that the initial tested condition is false. Whilst the test of a specific configuration in the real Bombe happens at the speed of light, the moving to the next wheel setting happens mechanically. Thus despite, the test being faster on the Bombe than on a modern computer, the modern computer can outperform the Bombe for the overall algorithm.

In other words the Bombe deduces a "Theorem" of the form

$$\text{If } (F \to H \text{ and the wheels are in position } \mathcal{P}) \text{ Then } K \to J.$$

With this interpretation the diagonal board detailed in descriptions of the Bombe is then the obvious condition that if K is plugged to J, then J is also plugged to K, i.e. if bit J of register K is set, then so must be bit K of register J. In the real Bombe this is achieved using wires, however in a computer simulation it means that we always set the "transpose" bit when setting any bit in our register bank. Thus, the register bank is symmetric down the leading diagonal. The diagonal board drastically increases the usefulness of the Bombe in breaking arbitrary cribs.

To understand how the menu acts on the set of registers we define the following permutation for $0 \leq i < 26^3$, for a given choice of rotors ρ_1, ρ_2 and ρ_3. We write $i = i_1 + i_2 \cdot 26 + i_3 \cdot 26^2$, and define

$$\begin{aligned}\delta_{i,s} &= (\sigma^{i_1+s+1} \cdot \rho_1 \cdot \sigma^{-i_1-s-1}) \cdot (\sigma^{i_2} \cdot \rho_2 \cdot \sigma^{-i_2}) \cdot (\sigma^{i_3} \cdot \rho_3 \cdot \sigma^{-i_3}) \cdot \varrho \\ &\quad \cdot (\sigma^{i_3} \cdot {\rho_3}^{-1} \cdot \sigma^{-i_3}) \cdot (\sigma^{i_2} \cdot {\rho_2}^{-1} \cdot \sigma^{-i_2}) \cdot (\sigma^{i_1+s+1} \cdot {\rho_1}^{-1} \cdot \sigma^{-i_1-s-1}).\end{aligned}$$

Note how similar this is to the equation of the Enigma machine. The main difference is that the second and third rotors cycle through at a different rate (depending only on i). The variable i is used to denote the rotor position which we currently with to test and the variable s is used to denote the action of the menu, as we shall now describe.

The menu acts on the registers as follows: for each link $x \xrightarrow{s} y$ in the menu we take register x and for each set bit x_z we apply $\delta_{i,s}$ to obtain x_w. Then the bit x_w is set in register y and (due to the diagonal board) bit y is set in register x_w. We also need to apply the link backwards, so for each set bit y_z in register y we apply $\delta_{i,s}$ to obtain y_w. Then bit y_w is set in register x and (again due to the diagonal board) bit x is set in register y_w.

We now let l denote the letter which satisfies at least one of the following, and we hope all three:

(1) A letter which occurs more often than any other letter in the menu.
(2) A letter which occurs in more cycles than any other letter.
(3) A letter which occurs in the largest connected component of the graph of the menu.

In the above two menus we several letters to choose from in Menu 1, so we select $l = S$; in Menu 2 we select $l = E$. For each value of i we then perform the following operation

- Unset all bits in the registers.
- Set bit l of register l.
- Keep applying the menu, as above, until the registers no longer change at all.

Hence, the above algorithm is working out the consequences of the letter l being plugged to itself, given the choice of rotors ρ_1, ρ_2 and ρ_3. It is the third line in the above algorithm which operates at the speed of light in the real Bombe. In a modern simulation this takes a lot longer.

After the registers converge to a steady state we then test them to see whether a possible value of i, i.e. a possible value of the rotor position has been found. We then step i on by one, which in the real Bombe is achieved by rotating the rotors, and repeat. A value of i which corresponds to a valid value of i is called a "Bombe Stop".

To see what is a valid value of i, suppose we have the rotors in the correct positions. If the plugboard hypothesis that the letter l is plugged to itself is true, then the registers will converge to a state which gives the plugboard settings for the registers in the graph of the menu which are connected to the letter l. If, however, the plugboard hypothesis is wrong then the registers will converge to a different state, in particular the bit of each register which corresponds to the correct plugboard configuration will never be set. The best we can then expect is that this wrong hypothesis propagates and all registers in the connected component become set with 25 bits. The one remaining unset bit then corresponds to the correct plugboard setting for the letter l. If the rotor position is wrong then it is highly likely that all the bits in the test register l converge to the set position.

To summarize, we have one of the following situations upon convergence of the registers at step i:

- All 26 bits of test register l are set. This implies that the rotors are not in the correct position and we can step on i by one and repeat the whole process.
- One bit of test register l is set, the rest being unset. This is a possible correct configuration for the rotors. If it is indeed the correct configuration then, in addition, the set bit corresponds to the correct plug setting for register l, and the single bit set in the registers

corresponding to the letters connected to l in the menu will give us the plug settings for those letters as well.

- One bit of the test register l is unset, the rest being set. This is also a possible correct configuration for the rotors. If it is indeed the correct configuration then, in addition, the unset bit corresponds to the correct plug setting for register l, and any single unset bit in the registers corresponding to the letters connected to l in the menu will give us the plug settings for those letters as well.
- The number of set bits in register l lies in $[2, \ldots, 24]$. These are relatively rare occurrences, and although they could correspond to actual rotor settings they tell us little directly about the plug settings. The problem could be because the initial plug hypothesis is false. For "good" menus we find that stops like this are very rare indeed.

A Bombe stop is a position where the machine decides that it has reached a possibly correct configuration of the rotors. The number of such stops per rotor order depends on the structure of the graph of the menu. Turing determined the expected number of stops for different types of menus. The following table shows the expected number of stops per rotor order for a connected menu (i.e. only one component) with various numbers of letters and cycles.

	Number of Letters								
Cycles	8	9	10	11	12	13	14	15	16
3	2.2	1.1	0.42	0.14	0.04	≈ 0	≈ 0	≈ 0	≈ 0
2	58	28	11	3.8	1.2	0.3	0.06	≈ 0	≈ 0
1	1500	720	280	100	31	7.7	1.6	0.28	0.04
0	40000	19000	7300	2700	820	200	43	7.3	1.0

This also gives an upper bound on the expected number of stops for an unconnected menu in terms of the size of the largest connected component and the number of cycles within the largest connected component.

Hence, a good menu is not only one which has a large connected component but which also has a number of cycles. Our second example menu is particularly poor in this respect. Note that a large number of letters in the connected component not only reduces the expected number of Bombe stops but also increases the number of deductions about possible plugboard configurations.

8.8.2. Bombe Stop to Plugboard: We now need to work out how from a Bombe stop we can either deduce the actual key, or deduce that the stop has occurred simply by chance and does not correspond to a correct configuration. We first sum up how many stops there are in our example above. For each menu we specify, in the following table, the number of Bombe stops which arise and we also specify the number of bits in the test register l which gave rise to the stop.

	Number of Bits Set									
Menu	1	2	3	4	5-20	21	22	23	24	25
1	137	0	0	0	0	0	0	0	9	1551
2	2606	148	9	2	0	2	7	122	2024	29142

Here we can see the effect of the difference in size of the largest connected component. In both menus the largest connected component has a single cycle in it; in both cases being an edge with two different labels. For the first menu we obtain a total of 1697 stops, or 28.3 stops per rotor order. The connected component has eleven letters in it, so this yield is much better than the yield expected from Turing's earlier table. This is due to the extra two-letter component in the graph of Menu One. For Menu Two we obtain a total of 34 062 stops, or 567.7 stops per rotor order. The connected component in the second menu has six letters in it, so although this figure is bad it is in

fact better than the maximum expected from Turing's table. Again this is due to the presence of other components in the graph.

Given the large number of stops we need a way of automating the checking process. It turns out that this is relatively simple as the state of the registers allow other conditions to be checked automatically. Recall that the Bombe stop also gives us information about the state of the supposed plugboard. The following are so-called "legal contradictions", which can be eliminated instantly from the above stops, assuming the initial plug supposition is correct:

- If *any* Bombe register has 26 bits set then this Bombe configuration is impossible.
- If the Bombe registers imply that a letter is plugged to two different letters then this is clearly a contradiction.

Suppose we know that the plugboard uses a certain number of plugs (in our example this number is ten); if the registers imply that there are more than this number of plugs then this is also a contradiction. Applying these conditions means we are down to only 19 750 possible Bombe stops out of the 35 759 total stops above. Of these, 109 correspond to the first menu and the rest correspond to the second menu.

We clearly cannot cope with all of those corresponding to the second menu so let's suppose that the second rotor does not turn over in the first thirteen characters. This means we now only need to focus on the first menu. In practice a number of configurations could be eliminated due to operational requirements imposed on the German operators (e.g. not using the same rotor order on consecutive days).

8.8.3. Finding the Final Part of the Key: We will focus on the first two remaining stops for the first menu. Both of these correspond to rotor orders where the rightmost (fastest) rotor is rotor I, the middle one is rotor II and the leftmost rotor is rotor III.

The first remaining stop is at Bombe configuration $i_1 = p_1 - r_1 = Y$, $i_2 = p_2 - r_2 = W$ and $i_3 = p_3 - r_3 = K$. These follow from the following final register state in this configuration given in Table 8.1, where rows represent registers and columns the bits. The test register S has 25 bits

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
B	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
C	0	0	0	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0	0
D	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
F	0	0	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1	1	0
G	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
H	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	1	1	0	1	0	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
K	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
L	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	0	1	1	1	1	0	0	0
M	0	0	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	0	0
N	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
O	0	1	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	1	1	1	1	0	1	1	1	1
P	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
Q	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
R	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
S	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
T	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
U	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1	1
V	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
W	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
X	0	0	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	0	1	1	1	1	0
Y	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0
Z	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	0	1	1	1	0	0	0

TABLE 8.1. The registers on the first Bombe stop

set, so in this configuration each bit of the test register implies that this letter is not plugged to another letter. The plugboard setting is deduced to contain the plugs

$$C \leftrightarrow D,\ E \leftrightarrow I,\ F \leftrightarrow N,\ H \leftrightarrow J,\ L \leftrightarrow S,$$
$$M \leftrightarrow R,\ O \leftrightarrow V,\ T \leftrightarrow Z,\ U \leftrightarrow W,$$

whilst the letter G is known to be plugged to itself, assuming this is the correct configuration.

So we need to find one other plug and the ring settings. We can assume that $r_3 = 0 = A$ as it plays no part in the actual decryption process. Since we are using the rotor I as the rightmost rotor we know that $n_1 = 16$, i.e. Q, which, combined with the fact that we are assuming that no stepping occurs in the first thirteen characters, implies that p_1 must satisfy

$$j - ((16 - p_1 - 1) \pmod{26}) + 26 \leq 25 \text{ for } j = 0, \ldots, 12.$$

i.e. $p_1 = 0, 1, 2, 16, 17, 18, 19, 20, 21, 22, 23, 24$ or 25.

With the Enigma setting of $p_1 = Y$, $p_2 = W$, $p_3 = K$ and $r_1 = r_2 = r_3 = A$ and the above (incomplete) plugboard we decrypt the fragment of ciphertext and compare the resulting plaintext with the crib.

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DVGGERLISHERGMWBRXZSWNVOMQOQKLKCSQLRRHPVCAG
DOGGERFISHERGERMANBIGHTEAST----------------
```

This is very much like the supposed plaintext. Examine the first incorrect letter, which occurs in position two. This error cannot be due to a second rotor turnover, because of our assumption, hence it must be due to a missing plugboard element. If we let γ_1 denote the current approximation to the permutation representing the Enigma machine for letter one and τ the missing plugboard setting then we have

$$U^{\gamma_1} = V \text{ and } U^{\tau \cdot \gamma_1 \cdot \tau} = O.$$

This implies that τ should contain either a plug involving the letter U or one involving the letter O; but both of these letters are already used in the plugboard output from the Bombe. Hence, this configuration must be incorrect.

The second remaining stop is at Bombe configuration $i_1 = p_1 - r_1 = R$, $i_2 = p_2 - r_2 = D$ and $i_3 = p_3 - r_3 = L$. The plugboard setting is deduced to contain the following plugs

$$D \leftrightarrow Q,\ E \leftrightarrow T,\ F \leftrightarrow N,\ I \leftrightarrow O,\ S \leftrightarrow V,\ W \leftrightarrow X,$$

whilst the letters G, H and R are known to be plugged to themselves, assuming this is the correct configuration. These follow from the final register state in this configuration given in Table 8.2. So we need to find four other plug settings and the ring settings. Again we can assume that $r_3 = A$ as it plays no part in the actual decryption process, and again we deduce that p_1 must be one of $0, 1, 2, 16, 17, 18, 19,\ 20, 21, 22, 23, 24$ or 25.

With the Enigma setting of $p_1 = R$, $p_2 = D$, $p_3 = L$ and $r_1 = r_2 = r_3 = A$ and the above (incomplete) plugboard we decrypt the fragment of ciphertext and compare the resulting plaintext with the crib.

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGNRAMNCOXHXZMORIKOEDEYWEFEYMSDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

We now look at the first incorrect letter, occurring in the 14th position. Using the same notation as before, i.e. γ_j for the current approximation and τ for the missing plugs, we see that if this incorrect operation is due to a plug problem rather than a rotor turnover problem then we must have

$$C^{\tau \cdot \gamma_{13} \cdot \tau} = E.$$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
B	0	0	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0	0
C	0	0	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
D	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
F	0	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0
G	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
H	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0	0
K	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
L	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
M	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
N	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
O	0	0	0	1	1	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	0	0	1
P	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0
Q	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0	0
R	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
S	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
T	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
U	0	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	0	0	1
V	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
W	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
X	0	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1	1	1	1	0	1	0	1	1	0
Y	0	0	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	1	1	1	0	0
Z	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0

TABLE 8.2. The registers on the second Bombe stop

Now, E already occurs on the plugboard, via $E \leftrightarrow T$, so τ must include a plug which maps C to the letter x where

$$x^{\gamma_{13}} = E.$$

But we can compute that

$$\gamma_{13} = (AM)(BE)(CN)(DO)(FI)(GS)(HX)(JU)(KP)(LQ)(RV)(TY)(WZ),$$

from which we deduce that $x = B$. So we include the plug $C \leftrightarrow B$ in our new approximation and repeat to obtain the plaintext

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERAMNBOXHXNMORIKOEMEYWEFEYMSDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

We then see in the 16th position that we either need to step the rotor or there should be a plug which means that S maps to M under the cipher. We have, for our new γ_{15} that

$$\gamma_{15} = (AS)(BJ)(CY)(DK)(EX)(FW)(GI)(HU)(LM)(NQ)(OP)(RV)(TZ).$$

The letter S already occurs in a plug, so we must have that A is plugged to M. We add this plug into our configuration and repeat

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBOXHXNAORIKVEAEYWEFEYASDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

Now the 20th character is incorrect: we need P to map to I and not O under the cipher in this position. Again assuming that this is due to a missing plug we find that

$$\gamma_{19} = (AH)(BM)(CF)(DY)(EV)(GX)(IK)(JR)(LS)(NT)(OP)(QW)(UZ).$$

There is already a plug involving the letter I so we deduce that the missing plug should be $K \leftrightarrow P$. Again we add this new plug into our configuration and repeat to obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIXHJNAORIPVXAEYWEFEYASDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

Now the 21st character is wrong as we must have that L maps to G. We know, from the Bombe stop configuration that G is plugged to itself, and given

$$\gamma_{20} = (AI)(BJ)(CW)(DE)(FK)(GZ)(HU)(LX)(MQ)(NT)(OV)(PY)(RS),$$

we deduce that if this error is due to a plug we must have that L is plugged to Z. We add this final plug into our configuration and find that we obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHJNAORIPVXAEYWEFEYAQDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

All the additional plugs we have added have been on the assumption that no rotor turnover has yet occurred. Any further errors must be due to rotor turnover, as we now have a full set of plugs (as we know our configuration only has ten plugs in use). If when correcting the rotor turnover we still do not decrypt correctly we need to back up and repeat the process.

We see that the next error occurs in position 23. This means that a rotor turnover must have occurred just before this letter was encrypted. In other words we have

$$22 - ((16 - p_1 - 1) \pmod{26}) + 26 = 26.$$

This implies that $p_1 = 19$, i.e. $p_1 = T$, which implies that $r_1 = C$. We now try to decrypt again, and we obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTZWORIPVXAEYWEFEYAQDQ
DOGGERFISHERGERMANBIGHTEAST----------------
```

But we still do not have the correct plaintext. The only thing which could have happened is that we have had an incorrect third rotor movement. Rotor II has its notch in position $n_2 = 4$, i.e. E. If the third rotor moved on at position 24 then we have, in our earlier notation

$$\begin{aligned} m_1 &= n_1 - p_1 - 1 \pmod{26} = 16 - 19 - 1 \pmod{26} = 22, \\ m &= n_2 - p_2 - 1 \pmod{26} = 4 - p_2 - 1 \pmod{26}, \\ m_2 &= m_1 + 1 + 26 \cdot m = 23 + 26 \cdot m \\ 650 &= 23 - m_2 + 650 \end{aligned}$$

This last equation implies that $m_2 = 23$, which implies that $m = 0$, which itself implies that $p_2 = 3$, i.e. $p_2 = D$. But this is exactly the setting we have for the second rotor. So the problem is not that the third rotor advances, it is that it should not have advanced. We therefore need to change this to say $p_2 = E$ and $r_2 = B$, (although this is probably incorrect it will help us to decrypt the fragment). We find that we then obtain

```
HUSVTNXRTSWESCGSGVXPLQKCEYUHYMPBNUITUIHNZRS
DOGGERFISHERGERMANBIGHTEASTFORCEFIVEFALLING
DOGGERFISHERGERMANBIGHTEAST----------------
```

Hence, we can conclude that, apart from a possibly incorrect setting for the second ring we have the correct Enigma setting for this day.

8.9. Ciphertext Only Attack

The following attack allows one to break the Enigma machine when only a single ciphertext is given. The method relies on the fact that enough ciphertext is given and that a full set of plugs is not used. Suppose we have a reasonably large amount of ciphertext, say 500-odd characters, and that p plugs are in use. If we knew the rotor positions, around $((26 - 2 \cdot p)/26)^2$ of the letters

would decrypt exactly, as these letters would not pass through a plug either before or after the rotor stage. Hence, one could distinguish the correct rotor positions by using some statistic to distinguish a random plaintext from a plaintext in which $((26-2\cdot p)/26)^2$ of the letters are correct.

Gillogly[2] suggests using the index of coincidence. To obtain this statistic we count the frequency f_i of each letter in the resulting plaintext of length n and compute

$$IC = \sum_{i=A}^{Z} \frac{f_i \cdot (f_i - 1)}{n \cdot (n-1)}.$$

For this approach we set the rings to position A, A, A and then run through all possible rotor orders and rotor starting positions. For each setting we compute the resulting plaintext and the associated value of IC. We keep those settings which have a high value of IC.

Gillogly then suggests for the settings which give a high value of IC, to run through the associated ring settings – adjusting the starting positions as necessary – with a similar test. The problem with this approach is that it is susceptible to the effect of turnover of the various rotors. Either a rotor could turn over when we did not expect it, or it could have turned over by error. This is similar to the situation we obtained in our example using the Bombe in a known plaintext attack.

Consider the following ciphertext, of 734 characters in length

```
RSDZANDHWQJPPKOKYANQIGTAHIKPDFHSAWXDPSXXZMMAUEVYYRLWVFFTSDYQPS
CXBLIVFDQRQDEBRAKIUVVYRVHGXUDNJTRVHKMZXPRDUEKRVYDFHXLNEMKDZEWV
OFKAOXDFDHACTVUOFLCSXAZDORGXMBVXYSJJNCYOHAVQYUVLEYJHKKTYALQOAJ
QWHYVVGLFQPTCDCAZXIZUOECCFYNRHLSTGJILZJZWNNBRBZJEEXAEATKGXMYJU
GHMCJRQUODOYMJCXBRJGRWLYRPQNABSKSVNVFGFOVPJCVTJPNFVWCFUUPTAXSR
VQDATYTTHVAWTQJPXLGBSIDWQNVHXCHEAMVWXKIUSLPXYSJDUQANWCBMZFSXWH
JGNWKIOKLOMNYDARREPGEZKCTZNPQKOMJZSQHYEADZTLUPGBAVCVNJHXQKYILX
LTHZXJKYFQEBDBQOHMXBTVXSRGMPVOGMVTEYOCQEOZUSLZDQZBCXXUXBZMZSWX
OCIWRVGLOEZWVVOQJXSFYKDQDXJZYNPGLWEEVZDOAKQOUOTUEBTCUTPYDHYRUS
AOYAVEBJVWGZHGLHBDHHRIVIAUUBHLSHNNNAZWYCCOFXNWXDLJMEFZRACAGBTG
NDIHOWFUOUHPJAHYZUGVJEYOBGZIOUNLPLNNZHFZDJCYLBKGQEWTQMXJKNYXPC
KAPJGAGKWUCLGTFKYFASCYGTXGZXXACCNRHSXTPYLSJWIEMSABFH
```

We run through all possible $60 \cdot 26^3$ possible values for the rotors and the rotor positions, with ring settings equal to A, A, A. We obtain the "high" values for the IC statistic given in Table 8.3. For the top 300 or so such high values we then run through all possible values for the rings r_1 and r_2 (note the third ring plays no part in the process) and we set the rotor starting positions to be

$$\begin{aligned} p_1 &= p_1' + r_1 + i_1, \\ p_2 &= p_2' + r_2 + i_2, \\ p_3 &= p_3' \end{aligned}$$

The addition of the r_j value is to take into account the change in ring position from A to r_j. The additional value of i_j is taken from the set $\{-1, 0, 1\}$ and is used to accommodate issues to do with rotor turnovers which our crude IC statistic is unable to pick up.

Running through all these possibilities we present the configurations producing the highest values of IC in Table 8.4. Finally, using our previous technique for finding the plugboard settings given the rotor settings in a ciphertext only attack (using the Sinkov statistic), we determine that the actual settings are

ρ_1	ρ_2	ρ_3	p_1	p_2	p_3	r_1	r_2	r_3
I	II	III	L	D	C	Q	B	A

[2]See the references at the end of this chapter.

IC	ρ_1	ρ_2	ρ_3	p'_1	p'_2	p'_3
0.04095	I	V	IV	P	R	G
0.0409017	IV	I	II	N	O	R
0.0409017	IV	V	I	M	G	Z
0.0408496	V	IV	II	I	J	B
0.040831	IV	I	V	X	D	A
0.0408087	II	I	V	E	O	J
0.040805	I	IV	III	T	Y	H
0.0407827	V	I	II	J	H	F
0.040779	III	IV	II	R	L	Q
0.0407121	II	III	V	V	C	C
0.0406824	IV	V	III	K	S	D
0.0406675	IV	II	III	H	H	D
0.04066	III	I	IV	P	L	G
0.0406526	IV	V	II	E	E	O
0.0406415	I	II	III	V	D	C
0.0406303	I	II	IV	T	C	G
0.0406266	V	IV	II	I	I	A
0.0406229	II	III	IV	K	Q	I
0.0405969	V	II	III	K	O	R
0.0405931	I	III	V	K	B	O
0.0405931	II	IV	I	K	B	Q
⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 8.3. High IC values for ring setting A, A, A

with the plugboard given by the following eight plugs:

$$A \leftrightarrow B,\ C \leftrightarrow D,\ E \leftrightarrow F,\ G \leftrightarrow H,$$

$$I \leftrightarrow J,\ K \leftrightarrow L,\ M \leftrightarrow N,\ O \leftrightarrow P.$$

With these settings one finds that the plaintext is again the first two paragraphs of "A Tale of Two Cities".

Chapter Summary

- We have described the Enigma machine and shown how poor session key agreement was used to break into the German traffic.
- We have also seen how stereotypical messages were successfully used to attack the system.
- We have seen how the plugboard and the rotors worked independently of each other, which led to attackers being able to break each component separately.

IC	ρ_1	ρ_2	ρ_3	p_1	p_2	p_3	r_1	r_2	r_3
0.0447751	I	II	III	K	D	C	P	B	A
0.0444963	I	II	III	L	D	C	Q	B	A
0.0444406	I	II	III	J	D	C	O	B	A
0.0443848	I	II	III	K	E	D	P	B	A
0.0443588	I	II	III	K	I	D	P	F	A
0.0443551	I	II	III	K	H	D	P	E	A
0.0443476	I	II	III	K	F	D	P	C	A
0.0442807	I	II	III	L	E	D	Q	B	A
0.0442324	I	II	III	J	H	D	O	E	A
0.0442064	I	II	III	K	G	D	P	D	A
0.0441357	I	II	III	J	G	D	O	D	A
0.0441097	I	II	III	J	E	D	O	B	A
0.0441097	I	II	III	L	F	D	Q	C	A
0.0441023	I	II	III	L	C	C	Q	A	A
0.0440837	I	II	III	J	F	D	O	C	A
0.0440763	I	II	III	J	I	D	O	F	A
0.0440242	I	II	III	K	C	C	P	A	A
0.0439833	I	II	III	L	G	D	Q	D	A
0.0438904	I	II	III	L	I	D	Q	F	A
0.0438607	I	II	III	L	H	D	Q	E	A
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 8.4. High IC values for different ring and rotor settings

Further Reading

The paper by Rejewski presents the work of the Polish cryptographers very clearly. The pure ciphertext only attack is presented in the papers by Gillogly and Williams.

J. Gillogly. *Ciphertext-only cryptanalysis of Enigma.* Cryptologia, **19**, 405–413, 1995.

M. Rejewski. *An application of the theory of permutations in breaking the Enigma cipher.* Applicationes Mathematicae, **16**, 543–559, 1980.

H. Williams. *Applying statistical language recognition techniques in the ciphertext-only cryptanalysis of Enigma.* Cryptologia, **24**, 4–17, 2000.

CHAPTER 9

Information-Theoretic Security

Chapter Goals

- To introduce the concept of perfect secrecy.
- To discuss the security of the one-time pad.
- To introduce the concept of entropy.
- To explain the notions of key equivocation, spurious keys and unicity distance.

9.1. Introduction

Information theory is one of the foundations of computer science. In this chapter we will examine its relationship to cryptography. But we shall not assume any prior familiarity with information theory.

We first need to overview the difference between information-theoretic security and computational security. Informally, a cryptographic system is called *computationally secure* if the best *possible* algorithm for breaking it requires N operations, where N is such a large number that it is infeasible to carry out this many operations. With current computing power we assume that 2^{128} operations is an infeasible number of operations to carry out. Hence, a value of N larger than 2^{128} would imply that the system is computationally secure. Note that no actual system can be proved secure under this definition, since we never know whether there is a better algorithm than the one known. Hence, in practice we say a system is computationally secure if the best *known* algorithm for breaking it requires an unreasonably large amount of computational resources.

Another practical approach, related to computational security, is to reduce breaking the system to solving some well-studied hard problem. For example, we can try to show that a given system is secure if a given integer N cannot be factored. Systems of this form are often called provably secure. However, we only have a proof relative to some hard problem, and hence this does not provide a complete guarantee of security.

Essentially, a computationally secure scheme, or one which is provably secure, is only secure when we consider an adversary whose computational resources are bounded. Even if the adversary has large, but limited, resources, she still will not break the system. When considering schemes which are computationally secure we need to be very clear about certain issues:

- We need to be careful about the key sizes etc. If the key size is small then our adversary may have enough computational resources to break the system.
- We need to keep abreast of current algorithmic developments and developments in computer hardware.
- At some point in the future we should expect our system to become broken, either through an improvement in computing power or an algorithmic breakthrough.

It turns out that most schemes in use today are computationally secure, and so every chapter in this book (except this one) will mainly focus on computationally secure systems.

On the other hand, a system is said to be *unconditionally secure* when we place no limit on the computational power of the adversary. In other words a system is unconditionally secure if it cannot be broken even with infinite computing power. Hence, no matter what algorithmic improvements are made or what improvements in computing technology occur, an unconditionally secure scheme will never be broken. Other names for unconditional security you find in the literature are perfect security or information-theoretic security.

You have already seen that the following systems are not computationally secure, since we already know how to break them with very limited computing resources:

- Shift cipher,
- Substitution cipher,
- Vigenère cipher,
- Enigma machine.

Of the systems we shall meet later, the following are computationally secure but are not unconditionally secure:

- DES and AES,
- RSA,
- ElGamal encryption.

However, the one-time pad which we shall meet in this chapter is unconditionally secure, but only if it is used correctly.

9.2. Probability and Ciphers

Before we can formally introduce the concept of unconditional security we first need to understand in more detail the role of probability in simple symmetric ciphers such as those discussed in Chapter 7. We utilize the following notation for various spaces:

- Let $\mathbb{P}$ denote the set of possible plaintexts.
- Let $\mathbb{K}$ denote the set of possible keys.
- Let $\mathbb{C}$ denote the set of ciphertexts.

To each of these sets we can assign a probability distribution, where we denote the probabilities by $p(P = m)$, $p(K = k)$, $p(C = c)$. We denote the encryption function by e_k, and the decryption function by d_k. For example, if our message space is $\mathbb{P} = \{a, b, c, d\}$ and the message a occurs with probability $1/4$ then we write

$$p(P = a) = \frac{1}{4}.$$

We make the reasonable assumption that P and K are independent, i.e. the user will not decide to encrypt certain messages under one key and other messages under another. The set of ciphertexts under a specific key k is defined by

$$\mathbb{C}(k) = \{e_k(x) : x \in \mathbb{P}\}.$$

We then have that $p(C = c)$ is defined by

$$p(C = c) = \sum_{k:c\in\mathbb{C}(k)} p(K = k) \cdot p(P = d_k(c)). \tag{9}$$

As an example, which we shall use throughout this section, assume that we have only four messages $\mathbb{P} = \{a, b, c, d\}$ which occur with probability

- $p(P = a) = 1/4$,
- $p(P = b) = 3/10$,
- $p(P = c) = 3/20$,
- $p(P = d) = 3/10$.

Also suppose we have three possible keys given by $\mathbb{K} = \{k_1, k_2, k_3\}$, which occur with probability

- $p(K = k_1) = 1/4$,
- $p(K = k_2) = 1/2$,
- $p(K = k_3) = 1/4$.

Now, suppose we have $\mathbb{C} = \{1, 2, 3, 4\}$, with the encryption function given by the following table.

	a	b	c	d
k_1	3	4	2	1
k_2	3	1	4	2
k_3	4	3	1	2

We can then compute, using formula (9),

$$\begin{aligned}
p(C = 1) &= p(K = k_1) \cdot p(P = d) + p(K = k_2) \cdot p(P = b) + p(K = k_3) \cdot p(P = c) \\
&= 0.2625, \\
p(C = 2) &= p(K = k_1) \cdot p(P = c) + p(K = k_2) \cdot p(P = d) + p(K = k_3) \cdot p(P = d) \\
&= 0.2625, \\
p(C = 3) &= p(K = k_1) \cdot p(P = a) + p(K = k_2) \cdot p(P = a) + p(K = k_3) \cdot p(P = b) \\
&= 0.2625, \\
p(C = 4) &= p(K = k_1) \cdot p(P = b) + p(K = k_2) \cdot p(P = c) + p(K = k_3) \cdot p(P = a) \\
&= 0.2125.
\end{aligned}$$

Hence, the ciphertexts produced are distributed almost uniformly. For $c \in \mathbb{C}$ and $m \in \mathbb{P}$ we can compute the conditional probability $p(C = c \mid P = m)$. This is the probability that c is the ciphertext given that m is the plaintext

$$p(C = c \mid P = m) \;=\; \sum_{k : m = d_k(c)} p(K = k).$$

This sum of probabilities is the sum over all keys k for which the decryption function on input of c will output m. For our prior example we can compute these probabilities as

$$\begin{aligned}
&p(C = 1 \mid P = a) = 0, \quad && p(C = 2 \mid P = a) = 0, \\
&p(C = 3 \mid P = a) = 0.75, \quad && p(C = 4 \mid P = a) = 0.25,
\end{aligned}$$

$$\begin{aligned}
&p(C = 1 \mid P = b) = 0.5, \quad && p(C = 2 \mid P = b) = 0, \\
&p(C = 3 \mid P = b) = 0.25, \quad && p(C = 4 \mid P = b) = 0.25,
\end{aligned}$$

$$\begin{aligned}
&p(C = 1 \mid P = c) = 0.25, \quad && p(C = 2 \mid P = c) = 0.25, \\
&p(C = 3 \mid P = c) = 0, \quad && p(C = 4 \mid P = c) = 0.5,
\end{aligned}$$

$$\begin{aligned}
&p(C = 1 \mid P = d) = 0.25, \quad && p(C = 2 \mid P = d) = 0.75, \\
&p(C = 3 \mid P = d) = 0, \quad && p(C = 4 \mid P = d) = 0.
\end{aligned}$$

However, when we try to break a cipher we want the conditional probability the other way around, i.e. we want to know the probability of a given message occurring given only the ciphertext. We can compute the probability of m being the plaintext given that c is the ciphertext via

$$p(P = m \mid C = c) = \frac{p(P = m) \cdot p(C = c \mid P = m)}{p(C = c)}.$$

This conditional probability can be computed by anyone who knows the encryption function and the probability distributions of K and P. Using these probabilities one may be able to deduce some information about the plaintext once the ciphertext is known.

Returning to our previous example we compute

$$\begin{aligned} &p(P=a \mid C=1)=0, && p(P=b \mid C=1)=0.571,\\ &p(P=c \mid C=1)=0.143, && p(P=d \mid C=1)=0.286, \end{aligned}$$

$$\begin{aligned} &p(P=a \mid C=2)=0, && p(P=b \mid C=2)=0,\\ &p(P=c \mid C=2)=0.143, && p(P=d \mid C=2)=0.857, \end{aligned}$$

$$\begin{aligned} &p(P=a \mid C=3)=0.714, && p(P=b \mid C=3)=0.286,\\ &p(P=c \mid C=3)=0, && p(P=d \mid C=3)=0, \end{aligned}$$

$$\begin{aligned} &p(P=a \mid C=4)=0.294, && p(P=b \mid C=4)=0.352,\\ &p(P=c \mid C=4)=0.352, && p(P=d \mid C=4)=0. \end{aligned}$$

Hence

- If we see the ciphertext 1 then we know the message is not equal to a. We also can guess that it is more likely to be b rather than c or d.
- If we see the ciphertext 2 then we know the message is not equal to a or b, and it is quite likely that the message is equal to d.
- If we see the ciphertext 3 then we know the message is not equal to c or d and there is a good chance that it is equal to a.
- If we see the ciphertext 4 then we know the message is not equal to d, but cannot really guess with confidence whether the message is a, b or c.

So in our previous example the ciphertext does reveal a lot of information about the plaintext. But this is exactly what we wish to avoid: We want the ciphertext to give no information about the plaintext. A system with this property, that the ciphertext reveals nothing about the plaintext, is said to be *perfectly secure*.

Definition 9.1 (Perfect Secrecy)**.** *A cryptosystem has perfect secrecy if*

$$p(P=m \mid C=c)=p(P=m)$$

for all plaintexts m and all ciphertexts c.

This means the probability that the plaintext is m, given that we know the ciphertext is c, is the same as the probability that it is m without seeing c. In other words knowing c reveals no information about m. Another way of describing perfect secrecy is the following.

Lemma 9.2. *A cryptosystem has perfect secrecy if $p(C=c \mid P=m)=p(C=c)$ for all m and c.*

Proof. This follows trivially from the definition

$$p(P=m \mid C=c)=\frac{p(P=m)p(C=c \mid P=m)}{p(C=c)}$$

and the fact that perfect secrecy means $p(P=m \mid C=c)=p(P=m)$. □

The first result about perfect security is as follows.

Lemma 9.3. *Assume the cryptosystem is perfectly secure, then*

$$\#\mathbb{K} \geq \#\mathbb{C} \geq \#\mathbb{P},$$

where

- $\#\mathbb{K}$ *denotes the size of the set of possible keys,*
- $\#\mathbb{C}$ *denotes the size of the set of possible ciphertexts,*
- $\#\mathbb{P}$ *denotes the size of the set of possible plaintexts.*

PROOF. First note that in any encryption scheme, we must have

$$\#\mathbb{C} \geq \#\mathbb{P}$$

since encryption must be an injective map; we have to be able to decrypt after all.

We assume that every ciphertext can occur, i.e. $p(C = c) > 0$ for all $c \in \mathbb{C}$, since if this does not hold then we can alter our definition of $\mathbb{C}$. Then for any message m and any ciphertext c we have

$$p(C = c \mid P = m) = p(C = c) > 0.$$

For each m, this means that for all c there must be a key k such that

$$e_k(m) = c.$$

Hence, $\#\mathbb{K} \geq \#\mathbb{C}$ as required. □

We now come to the main theorem on perfectly secure ciphers, due to Shannon. Shannon's Theorem tells us exactly which encryption schemes are perfectly secure and which are not.

Theorem 9.4 (Shannon)**.** *Let* $(\mathbb{P}, \mathbb{C}, \mathbb{K}, e_k(\cdot), d_k(\cdot))$ *denote a cryptosystem with* $\#\mathbb{P} = \#\mathbb{C} = \#\mathbb{K}$*. Then the cryptosystem provides perfect secrecy if and only if*

- *Every key is used with equal probability* $1/\#\mathbb{K}$*,*
- *For each* $m \in \mathbb{P}$ *and* $c \in \mathbb{C}$ *there is a unique key* k *such that* $e_k(m) = c$*.*

PROOF. Note the statement is *if and only if*; hence we need to prove it in both directions. We first prove the *only if* part.

Suppose the system gives perfect secrecy. Then we have already seen, in the proof of Lemma 9.3, that for all $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is a key k such that $e_k(m) = c$. Now, since we have assumed $\#\mathbb{C} = \#\mathbb{K}$ we have

$$\#\{e_k(m) : k \in \mathbb{K}\} = \#\mathbb{K}$$

i.e. there do not exist two keys k_1 and k_2 such that

$$e_{k_1}(m) = e_{k_2}(m) = c.$$

So for all $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is exactly one $k \in \mathbb{K}$ such that $e_k(m) = c$. We need to show that every key is used with equal probability, i.e. $p(K = k) = 1/\#\mathbb{K}$ for all $k \in \mathbb{K}$.

Let $n = \#\mathbb{K}$ and $\mathbb{P} = \{m_i : 1 \leq i \leq n\}$, fix $c \in \mathbb{C}$ and label the keys $k_1, \ldots, k_n$ such that $e_{k_i}(m_i) = c$ for $1 \leq i \leq n$. We then have, noting that due to perfect secrecy $p(P = m_i \mid C = c) = p(P = m_i)$,

$$\begin{aligned} p(P = m_i) &= p(P = m_i \mid C = c) \\ &= \frac{p(C = c \mid P = m_i) \cdot p(P = m_i)}{p(C = c)} \\ &= \frac{p(K = k_i) \cdot p(P = m_i)}{p(C = c)}. \end{aligned}$$

Hence we obtain, for all $1 \leq i \leq n$, that $p(C = c) = p(K = k_i)$. This says that the keys are used with equal probability and hence $p(K = k) = 1/\#\mathbb{K}$ for all $k \in \mathbb{K}$.

Now we need to prove the result in the other direction. Namely, if

- $\#\mathbb{K} = \#\mathbb{C} = \#\mathbb{P}$,
- Every key is used with equal probability $1/\#\mathbb{K}$,

- For each $m \in \mathbb{P}$ and $c \in \mathbb{C}$ there is a unique key k such that $e_k(m) = c$,

then we need to show the system is perfectly secure, i.e. for all m and c that

$$p(P = m \mid C = c) = p(P = m).$$

Since each key is used with equal probability, we have

$$\begin{aligned} p(C = c) &= \sum_k p(K = k) \cdot p(P = d_k(c)) \\ &= \frac{1}{\#\mathbb{K}} \sum_k p(P = d_k(c)). \end{aligned}$$

Also, since for each m and c there is a unique key k with $e_k(m) = c$, we must have

$$\sum_k p(P = d_k(c)) = \sum_m p(P = m) = 1.$$

Hence, $p(C = c) = 1/\#\mathbb{K}$. In addition, if $c = e_k(m)$ then $p(C = c \mid P = m) = p(K = k) = 1/\#\mathbb{K}$. So using Bayes' Theorem we have

$$\begin{aligned} p(P = m \mid C = c) &= \frac{p(P = m) \cdot p(C = c \mid P = m)}{p(C = c)} \\ &= \frac{p(P = m) \cdot \frac{1}{\#\mathbb{K}}}{\frac{1}{\#\mathbb{K}}} \\ &= p(P = m). \end{aligned}$$

□

We end this section by discussing a couple of systems which have perfect secrecy.

9.2.1. Modified Shift Cipher: Recall that the shift cipher is one in which we "add" a given letter (the key) to each letter of the plaintext to obtain the ciphertext. We now modify this cipher by using a different key for each plaintext letter. For example, to encrypt the message HELLO we choose five random keys, say FUIAT. We then add the key to the plaintext, modulo 26, to obtain the ciphertext MYTLH. Notice how the plaintext letter L encrypts to different letters in the ciphertext.

When we use the shift cipher with a different random key for each letter, we obtain a perfectly secure system. To see why this is so, consider the situation of encrypting a message of length n. Then the total number of keys, ciphertexts and plaintexts are all equal, namely:

$$\#\mathbb{K} = \#\mathbb{C} = \#\mathbb{P} = 26^n.$$

In addition each key will occur with equal probability:

$$p(K = k) = \frac{1}{26^n},$$

and for each m and c there is a unique k such that $e_k(m) = c$. Hence, by Shannon's Theorem this modified shift cipher is perfectly secure.

9.2.2. Vernam Cipher: The above modified shift cipher basically uses addition modulo 26. One problem with this is that in a computer, or any electrical device, mod 26 arithmetic is hard, but binary arithmetic is easy. We are particularly interested in the addition operation, which is denoted by $\oplus$ and is equal to the logical exclusive-or operation.

$\oplus$	0	1
0	0	1
1	1	0

In 1917 Gilbert Vernam patented a cipher which used these principles, called the *Vernam cipher* or *one-time pad*. To send a binary string we need a key, which is a binary string as long as the message. To encrypt a message we exclusive-or each bit of the plaintext with each bit of the key to produce the ciphertext.

Each key is only allowed to be used once, hence the term *one-time* pad. This means that key distribution is a problem, which we shall come back to again and again. To see why we cannot get away with using a key twice, consider the following chosen plaintext attack. We assume that Alice always uses the same key k to encrypt a message to Bob. Eve wishes to determine this key and so carries out the following attack:

- Eve generates m and asks Alice to encrypt it.
- Eve obtains $c = m \oplus k$.
- Eve now computes $k = c \oplus m$.

You may object to this attack since it requires Alice to be particularly stupid, in that she encrypts a message for Eve. But in designing our cryptosystems we should try to make systems which are secure even against stupid users.

Another problem with using the same key twice is the following. Suppose Eve can intercept two messages encrypted with the same key

$$c_1 = m_1 \oplus k,$$
$$c_2 = m_2 \oplus k.$$

Eve can now determine some partial information about the pair of messages m_1 and m_2 since she can compute

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2.$$

Despite the problems associated with key distribution, the one-time pad has been used in the past in military and diplomatic contexts.

9.3. Entropy

If every message we send requires a key as long as the message, and we never encrypt two messages with the same key, then encryption will not be very useful in everyday applications such as Internet transactions. This is because getting the key from one person to another will be an impractical task. After all, one cannot encrypt it since that would require another key. This problem is called the key distribution problem.

To simplify the key distribution problem we need to turn from perfectly secure encryption algorithms to ones which are, we hope, computationally secure. This is the goal of modern cryptography, where one aims to build systems such that

- one key can be used many times,
- a small key can encrypt a long message.

Such systems will not be unconditionally secure, by Shannon's Theorem, and so must be at best only computationally secure.

We now need to develop the information theory needed to deal with these computationally secure systems. Again the main results are due to Shannon in the late 1940s. In particular, we shall use Shannon's idea of using *entropy* as a way of measuring information.

The word entropy is another name for uncertainty, and the basic tenet of information theory is that uncertainty and information are essentially the same thing. This takes some getting used to, but consider that if you are uncertain what something means then revealing the meaning gives you information. As a cryptographic application, suppose you want to determine the information in a ciphertext, in other words you want to know what the ciphertext's true meaning is. The entropy in the ciphertext is the amount of uncertainty you have about the underlying plaintext. If X is a random variable, the amount of entropy (in bits) associated with X is denoted by $H(X)$. We

shall define this quantity formally in a second. First, let us look at a simple example to help clarify ideas.

Suppose X is the answer to some question, i.e. *Yes* or *No*. If you know I will always say *Yes*, then my answer gives you no information. So the information contained in X should be zero, i.e. $H(X) = 0$. There is no uncertainty about what I will say, hence no information is given by me saying it, hence there is no entropy. On the other hand, if you have no idea what I will say and I reply *Yes* with equal probability to replying *No* then I am revealing one bit of information. Hence, we should have $H(X) = 1$.

Note that the entropy does not depend on the length of the actual message; in the above case we have a message of length at most three letters but the amount of information is at most one bit. We can now define formally the notion of entropy.

Definition 9.5 (Entropy)**.** *Let X be a random variable which takes a finite set of values x_i, with $1 \leq i \leq n$, and has probability distribution $p_i = p(X = x_i)$, where we use the convention that if $p_i = 0$ then $p_i \log_2 p_i = 0$. The entropy of X is defined to be*

$$H(X) = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i.$$

Let us return to our *Yes* or *No* question above and show that this definition of entropy coincides with our intuition. Recall that X is the answer to some question with responses *Yes* or *No*. If you know I will always say *Yes* then $p_1 = 1$ and $p_2 = 0$. We compute $H(X) = -1 \cdot \log_2 1 - 0 \cdot \log_2 0 = 0$. Hence, my answer reveals no information to you. If you have no idea what I will say and I reply *Yes* with equal probability to replying *No* then $p_1 = p_2 = 1/2$. We now compute

$$H(X) = -\frac{\log_2 \frac{1}{2}}{2} - \frac{\log_2 \frac{1}{2}}{2} = 1.$$

Hence, my answer reveals one bit of information to you.

9.3.1. Properties of Entropy: A number of elementary properties of entropy follow immediately from the definition.

- We always have $H(X) \geq 0$.
- The only way to obtain $H(X) = 0$ is if for some i we have $p_i = 1$ and $p_j = 0$ when $i \neq j$.
- If $p_i = 1/n$ for all i then $H(X) = \log_2 n$.

Another way of looking at entropy is that it measures how much one can compress information. If I send a single ASCII character to signal *Yes* or *No*, for example I could simply send Y or N, I am actually sending eight bits of data, but I am only sending one bit of information. If I wanted to I could compress the data down to 1/8th of its original size. Hence, naively if a message of length n can be compressed to a proportion ϵ of its original size then it contains $\epsilon \cdot n$ bits of information in it.

We now derive an upper bound for the entropy of a random variable, to go with our lower bound of $H(X) \geq 0$. To do this we will need the following special case of Jensen's inequality.

Theorem 9.6 (Jensen's Inequality)**.** *Suppose*

$$\sum_{i=1}^{n} a_i = 1$$

with $a_i > 0$ for $1 \leq i \leq n$. Then, for $x_i > 0$,

$$\sum_{i=1}^{n} a_i \cdot \log_2 x_i \leq \log_2 \left(\sum_{i=1}^{n} a_i \cdot x_i \right).$$

Equality occurs if and only if $x_1 = x_2 = \ldots = x_n$.

Using this we can now prove the following theorem.

Theorem 9.7. *If X is a random variable which takes n possible values then*

$$0 \leq H(X) \leq \log_2 n.$$

The lower bound is obtained if one value occurs with probability one, and the upper bound is obtained if all values are equally likely.

PROOF. We have already discussed the facts about the lower bound so we will concentrate on the statements about the upper bound. The hypothesis is that X is a random variable with probability distribution $p_1, \ldots, p_n$, with $p_i > 0$ for all i. One can then deduce the following sequence of inequalities:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{n} p_i \cdot \log_2 p_i \\ &= \sum_{i=1}^{n} p_i \cdot \log_2 \frac{1}{p_i} \\ &\leq \log_2 \left(\sum_{i=1}^{n} \left(p_i \cdot \frac{1}{p_i} \right) \right) \qquad \text{by Jensen's inequality} \\ &= \log_2 n. \end{aligned}$$

To obtain equality, we require equality when we apply Jensen's inequality. But this will only occur when $p_i = 1/n$ for all i, in other words, when all values of X are equally likely. □

9.3.2. Joint and Conditional Entropy: The basics of the theory of entropy closely match those of the theory of probability. For example, if X and Y are random variables then we define the joint probability distribution as

$$r_{i,j} = p(X = x_i \text{ and } Y = y_j)$$

for $1 \leq i \leq n$ and $1 \leq j \leq m$. The joint entropy is then obviously defined as

$$H(X, Y) = -\sum_{i=1}^{n} \sum_{j=1}^{m} r_{i,j} \cdot \log_2 r_{i,j}.$$

You should think of the joint entropy $H(X, Y)$ as the total amount of information contained in one observation of $(x, y) \in X \times Y$. We then obtain the inequality

$$H(X, Y) \leq H(X) + H(Y)$$

with equality if and only if X and Y are independent. We leave the proof of this as an exercise.

Just as with probability theory, where one has the linked concepts of joint probability and conditional probability, so the concept of joint entropy is linked to the concept of conditional entropy. This is important to understand, since conditional entropy is the main tool we shall use in understanding non-perfect ciphers in the rest of this chapter. Let X and Y be two random variables. Recall we defined the conditional probability distribution as

$$p(X = x \mid Y = y) = \text{ Probability that } X = x \text{ given } Y = y.$$

The entropy of X given an observation of $Y = y$ is then defined in the obvious way by

$$H(X \mid Y = y) = -\sum_{x} p(X = x \mid Y = y) \cdot \log_2 p(X = x \mid Y = y).$$

Given this, we define the conditional entropy of X given Y as

$$\begin{aligned} H(X \mid Y) &= \sum_y p(Y=y) \cdot H(X \mid Y=y) \\ &= -\sum_x \sum_y p(Y=y) \cdot p(X=x \mid Y=y) \cdot \log_2 p(X=x \mid Y=y). \end{aligned}$$

This is the amount of uncertainty about X that is left after revealing a value of Y. The conditional and joint entropy are linked by the following formula

$$H(X,Y) = H(Y) + H(X \mid Y)$$

and we have the following upper bound

$$H(X \mid Y) \leq H(X)$$

with equality if and only if X and Y are independent. Again, we leave the proof of these statements as an exercise.

9.3.3. Application to Ciphers: Now turning to cryptography again, we have some trivial statements relating the entropy of P, K and C.

- $H(P \mid K, C) = 0$:
  If you know the ciphertext and the key then you know the plaintext. This must hold since otherwise decryption will not work correctly.
- $H(C \mid P, K) = 0$:
  If you know the plaintext and the key then you know the ciphertext. This holds for all ciphers we have seen so far, and holds for all the block ciphers we shall see in later chapters. However, for modern encryption schemes we do not have this last property when they are used correctly, as many ciphertexts can correspond to the same plaintext.

In addition we have the following identities

$$\begin{aligned} H(K,P,C) &= H(P,K) + H(C \mid P,K) && \text{as } H(X,Y) = H(Y) + H(X \mid Y) \\ &= H(P,K) && \text{as } H(C \mid P,K) = 0 \\ &= H(K) + H(P) && \text{as } K \text{ and } P \text{ are independent} \end{aligned}$$

and

$$\begin{aligned} H(K,P,C) &= H(K,C) + H(P \mid K,C) && \text{as } H(X,Y) = H(Y) + H(X \mid Y) \\ &= H(K,C) && \text{as } H(P \mid K,C) = 0. \end{aligned}$$

Hence, we obtain

$$H(K,C) = H(K) + H(P).$$

This last equality is important since it is related to the conditional entropy $H(K \mid C)$, which is called the *key equivocation*. The key equivocation is the amount of uncertainty left about the key after one ciphertext is revealed. Recall that our goal is to determine the key given the ciphertext. Putting two of our prior equalities together we find

$$H(K \mid C) = H(K,C) - H(C) = H(K) + H(P) - H(C). \tag{10}$$

In other words, the uncertainty about the key left after we reveal a ciphertext is equal to the uncertainty in the plaintext and the key minus the uncertainty in the ciphertext.

Let us return to our baby cryptosystem considered in the previous section. Recall we had the probability spaces

$$\mathbb{P} = \{a, b, c, d\}, \quad \mathbb{K} = \{k_1, k_2, k_3\} \text{ and } \mathbb{C} = \{1, 2, 3, 4\},$$

with the associated probabilities:

- $p(P=a) = 0.25$, $p(P=b) = p(P=d) = 0.3$ and $p(P=c) = 0.15$,

- $p(K = k_1) = p(K = k_3) = 0.25$ and $p(K = k_2) = 0.5$,
- $p(C = 1) = p(C = 2) = p(C = 3) = 0.2625$ and $p(C = 4) = 0.2125$.

We can then calculate the relevant entropies as:

$$H(P) \approx 1.9527,$$
$$H(K) \approx 1.5,$$
$$H(C) \approx 1.9944.$$

Hence

$$H(K \mid C) \approx 1.9527 + 1.5 - 1.9944 \approx 1.4583.$$

So around one and a half bits of information about the key are left to be found, on average, after a single ciphertext is observed. This explains why the system leaks information, and shows that it cannot be secure. After all there are only 1.5 bits of uncertainty about the key to start with; one ciphertext leaves us with 1.4593 bits of uncertainty. Hence, $1.5 - 1.4593 = 0.042$ bits of information about the key are revealed by a single ciphertext, or equivalently three percent of the key.

9.4. Spurious Keys and Unicity Distance

In our baby example above, information about the key is leaked by an individual ciphertext, since knowing the ciphertext rules out a certain subset of the keys. Of the remaining possible keys, only one is correct. The remaining possible, but incorrect, keys are called the *spurious keys*.

Consider the (unmodified) shift cipher, i.e. where the same key is used for each letter. Suppose the ciphertext is WNAJW, and suppose we know that the plaintext is an English word. The only "meaningful" plaintexts are RIVER and ARENA, which correspond to the two possible keys F and W. One of these keys is the correct one and one is spurious.

We can now explain why it was easy to break the substitution cipher in terms of a concept called the *unicity distance* of the cipher. We shall explain this relationship in more detail, but we first need to understand the underlying plaintext in more detail. The plaintext in many computer communications can be considered as a random bit string. But often this is not so. Sometimes one is encrypting an image or sometimes one is encrypting plain English text. In our discussion we shall consider the case when the underlying plaintext is taken from English, as in the substitution cipher. Such a language is called a *natural language* to distinguish it from the bit streams used by computers to communicate.

We first wish to define the entropy (or information) per letter H_L of a natural language such as English. Note that a random string of alphabetic characters would have entropy

$$\log_2 26 \approx 4.70.$$

So we have $H_L \leq 4.70$. If we let P denote the random variable of letters in the English language then we have

$$p(P = a) = 0.082, \ \ldots, \ p(P = e) = 0.127, \ \ldots, \ p(P = z) = 0.001.$$

We can then compute

$$H_L \leq H(P) \approx 4.14.$$

Hence, instead of 4.7 bits of information per letter, if we only examine the letter frequencies we conclude that English conveys around 4.14 bits of information per letter.

But this is a gross overestimate, since letters are not independent. For example Q is almost always followed by U and the bigram TH is likely to be very common. One would suspect that a better statistic for the amount of entropy per letter could be obtained by looking at the distribution of bigrams. Hence, we let P^2 denote the random variable of bigrams. If we let $p(P = i, P' = j)$

denote the random variable which is assigned the probability that the bigram "ij" appears, then we define

$$H(P^2) = -\sum_{i,j} p(P = i, P' = j) \cdot \log p(P = i, P' = j).$$

A number of people have computed values of $H(P^2)$ and it is commonly accepted to be given by

$$H(P^2) \approx 7.12.$$

We want the entropy per letter so we compute

$$H_L \leq H(P^2)/2 \approx 3.56.$$

But again this is an overestimate, since we have not taken into account that the most common trigram is *THE*. Hence, we can also look at P^3 and compute $H(P^3)/3$. This will also be an overestimate, and so on,... This leads us to the following definition.

Definition 9.8. *The entropy of the natural language L is defined to be*

$$H_L = \lim_{n \longrightarrow \infty} \frac{H(P^n)}{n}.$$

The exact value of H_L is hard to compute exactly but we can approximate it. In fact one has, by experiment, that for English

$$1.0 \leq H_L \leq 1.5.$$

So each letter in English

- requires $5 = \lceil \log_2(26) \rceil$ bits of data to represent it,
- only gives at most 1.5 bits of information.

This shows that English contains a high degree of redundancy, in that there is far less information conveyed in an English sentence than the amount of data needed to represent the sentence. One can see this from the following, which you can still hopefully read (just) even though I have deleted two out of every four letters,

On** up** a t**e t**re **s a **rl **ll** Sn** Wh**e.

The *redundancy* of a language is defined by

$$R_L = 1 - \frac{H_L}{\log_2 \#\mathbb{P}},$$

and it expresses the percentage of text in the language which can be removed (in principle) without affecting the overall meaning. If we take $H_L \approx 1.25$ then the redundancy of English is

$$R_L \approx 1 - \frac{1.25}{\log_2 26} = 0.75.$$

So this means that we should be able to compress an English text file of around 10 MB down to 2.5 MB.

9.4.1. Redundancy and Ciphertexts: We now return to a general cipher and suppose $c \in \mathbb{C}^n$, i.e. c is a ciphertext consisting of n characters. We define $\mathbb{K}(c)$ to be the set of keys which produce a "meaningful" decryption of c. Then, clearly $\#\mathbb{K}(c) - 1$ is the number of spurious keys given c.

The average number of spurious keys is defined to be $\overline{s}_n$, where

$$\begin{aligned}\overline{s}_n &= \sum_{c\in\mathbb{C}^n} p(C=c)\cdot(\#\mathbb{K}(c)-1)\\ &= \sum_{c\in\mathbb{C}^n} p(C=c)\cdot\#\mathbb{K}(c) - \sum_{c\in\mathbb{C}^n} p(C=c)\\ &= \left(\sum_{c\in\mathbb{C}^n} \#\mathbb{K}(c)\cdot p(C=c)\right) - 1.\end{aligned}$$

Now if n is sufficiently large and $\#\mathbb{P} = \#\mathbb{C}$ we obtain

$$\begin{aligned}\log_2(\overline{s}_n+1) &= \log_2\left(\sum_{c\in\mathbb{C}^n} \#\mathbb{K}(c)\cdot p(C=c)\right)\\ &\geq \sum_{c\in\mathbb{C}^n} p(C=c)\cdot\log_2\#\mathbb{K}(c) && \text{by Jensen's inequality}\\ &\geq \sum_{c\in\mathbb{C}^n} p(C=c)\cdot H(K\mid c)\\ &= H(K\mid C^n) && \text{by definition}\\ &= H(K)+H(P^n)-H(C^n) && \text{equation (10)}\\ &\approx H(K)+n\cdot H_L - H(C^n) && \text{if } n \text{ is very large}\\ &= H(K)-H(C^n)\\ &\qquad + n\cdot(1-R_L)\cdot\log_2\#\mathbb{P} && \text{by definition of } R_L\\ &\geq H(K)-n\cdot\log_2\#\mathbb{C}\\ &\qquad + n\cdot(1-R_L)\cdot\log_2\#\mathbb{P} && \text{as } H(C^n)\leq n\cdot\log_2\#\mathbb{C}\\ &= H(K)-n\cdot R_L\cdot\log_2\#\mathbb{P} && \text{as } \#\mathbb{P}=\#\mathbb{C}.\end{aligned}$$

So, if n is sufficiently large and $\#\mathbb{P} = \#\mathbb{C}$ then

$$\overline{s}_n \geq \frac{\#\mathbb{K}}{\#\mathbb{P}^{n\cdot R_L}} - 1.$$

As an attacker we would like the number of spurious keys to become zero, and it is clear that as we take longer and longer ciphertexts then the number of spurious keys must go down.

The unicity distance n_0 of a cipher is the value of n for which the expected number of spurious keys becomes zero. In other words this is the average amount of ciphertext needed before an attacker can determine the key, assuming the attacker has infinite computing power. For a perfect cipher we have $n_0 = \infty$, but for other ciphers the value of n_0 can be alarmingly small. We can obtain an estimate of n_0 by setting $\overline{s}_n = 0$ in

$$\overline{s}_n \geq \frac{\#\mathbb{K}}{\#\mathbb{P}^{n\cdot R_L}} - 1$$

to obtain

$$n_0 \approx \frac{\log_2\#\mathbb{K}}{R_L\cdot\log_2\#\mathbb{P}}.$$

In the substitution cipher we have

$$\begin{aligned}\#\mathbb{P} &= 26,\\ \#\mathbb{K} &= 26! \approx 4\cdot 10^{26}\end{aligned}$$

and using our value of $R_L = 0.75$ for English we can approximate the unicity distance as

$$n_0 \approx \frac{88.4}{0.75 \times 4.7} \approx 25.$$

So we require on average only 25 ciphertext characters before we can break the substitution cipher, again assuming infinite computing power. In any case after 25 characters we expect a unique valid decryption.

Now assume we have a modern cipher which encrypts bit strings using keys of bit length l. We have

$$\#\mathbb{P} = 2,$$
$$\#\mathbb{K} = 2^l.$$

Again we assume $R_L = 0.75$, which is an underestimate since we now need to encode English into a computer communications medium such as ASCII. Then the unicity distance is

$$n_0 \approx \frac{l}{0.75} = \frac{4 \cdot l}{3}.$$

Now assume instead of transmitting the plain ASCII we compress it first. If we assume a perfect compression algorithm then the plaintext will have no redundancy and so $R_L \approx 0$. In which case the unicity distance is

$$n_0 \approx \frac{l}{0} = \infty.$$

So you may ask whether modern ciphers encrypt plaintexts with no redundancy? The answer is no; even if one compresses the data, a modern cipher often adds some redundancy to the plaintext before encryption. The reason is that we have only considered passive attacks, i.e. an attacker has been only allowed to examine ciphertexts and from these ciphertexts the attacker's goal is to determine the key. There are other types of attack called active attacks; in these an attacker is allowed to generate plaintexts or ciphertexts of her choosing and ask the key holder to encrypt or decrypt them, the two variants being called a chosen plaintext attack and a chosen ciphertext attack respectively. In public key systems that we shall see later, chosen plaintext attacks cannot be stopped since anyone is allowed to encrypt anything.

We would like to stop chosen ciphertext attacks for all types of cipher. The current wisdom for encryption algorithms is to make the cipher add some redundancy to the plaintext before it is encrypted. In this way it is hard for an attacker to produce a ciphertext which has a valid decryption. The philosophy is that it is then hard for an attacker to mount a chosen ciphertext attack, since it will be hard for an attacker to choose a valid ciphertext for a decryption query. We shall discuss this more in later chapters.

Chapter Summary

- A cryptographic system for which knowing the ciphertext reveals no more information than if you did not know the ciphertext is called a perfectly secure system.
- Perfectly secure systems exist, but they require keys as long as the message and a different key to be used with each new encryption. Hence, perfectly secure systems are not very practical.
- Information and uncertainty are essentially the same thing.
- The amount of uncertainty in a random variable is measured by its entropy.
- An attacker really wants, given the ciphertext, to determine some information about the plaintext.

- The equation $H(K \mid C) = H(K) + H(P) - H(C)$ allows us to estimate how much uncertainty remains about the key after one observes a single ciphertext.
- The natural redundancy of English means that a naive cipher does not need to produce a lot of ciphertext before the underlying plaintext can be discovered.

Further Reading

Our discussion of Shannon's theory has closely followed the treatment in the book by Stinson. Another possible source of information is the book by Welsh. A general introduction to information theory, including its application to coding theory, is in the book by van der Lubbe.

J.C.A. van der Lubbe. *Information Theory.* Cambridge University Press, 1997.

D. Stinson. *Cryptography: Theory and Practice.* Third Edition. CRC Press, 2005.

D. Welsh. *Codes and Cryptography.* Oxford University Press, 1988.

CHAPTER 10

Historical Stream Ciphers

Chapter Goals

- To introduce the general model of symmetric ciphers.
- To explain the relation between stream ciphers and the Vernam cipher.
- To examine the working and breaking of the Lorenz cipher in detail.

10.1. Introduction to Symmetric Ciphers

A symmetric cipher works using the following two transformations

$$c = e_k(m),$$
$$m = d_k(c)$$

where

- m is the plaintext,
- e is the encryption function,
- d is the decryption function,
- k is the secret key,
- c is the ciphertext.

It is desirable that both the encryption and decryption functions be public knowledge and so the secrecy of the message, given the ciphertext, depends totally on the secrecy of the secret key k. Although this well-established principle, called Kerckhoffs' principle, has been known since the mid-1800s some companies still ignore it and choose to deploy secret proprietary encryption schemes which usually turn out to be insecure as soon as someone leaks the details of the algorithms. The best schemes will be the ones which have been studied by many people for a very long time and which have been found to remain secure. A scheme which is a commercial secret cannot be studied by anyone outside the company.

The above set-up is called a symmetric key system since both parties need access to the secret key. Sometimes symmetric key cryptography is implemented using two keys, one for encryption and one for decryption. However, if this is the case we assume that, given the encryption key, it is easy to compute the decryption key (and vice versa). Later we shall meet public key cryptography where only one key is kept secret, called the private key; the other key, called the public key, is allowed to be published in the clear. In this situation it is assumed to be computationally infeasible for someone to compute the private key given the public key.

Returning to symmetric cryptography, a moment's thought reveals that the number of possible keys must be very large. This is because in designing a cipher we assume the worst-case scenario and give the attacker the benefit of

- full knowledge of the encryption/decryption algorithm,
- a number of plaintext/ciphertext pairs associated with the target key k.

If the number of possible keys is small then an attacker can break the system using an exhaustive search. The attacker encrypts one of the given plaintexts under all possible keys and determines which key produces the given ciphertext. Hence, the key space needs to be large enough to avoid such an attack. It is commonly assumed that a computation taking 2^{128} steps will be infeasible for a number of years to come, hence the key space size should be at least 128 bits to avoid exhaustive search.

The cipher designer must play two roles, that of someone trying to break a cipher, as well as someone trying to create one. These days, although there is a lot of theory behind the design of ciphers, we still rely on symmetric ciphers which are just believed to be strong, rather than ones for which we know a reason why they are strong. All this means is that the best attempts of the most experienced cryptanalysts cannot break them. This should be compared with public key ciphers and modes of operations of block ciphers, where there is now a theory which allows us to reason about how strong a given cipher is (given some explicit computational assumption on the underlying primitive).

Figure 10.1 describes a simple model for enciphering bits which, although simple, is quite suited to practical implementations. The idea of this model is to apply a reversible operation to

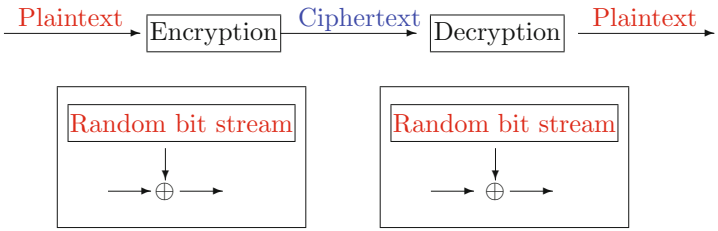


FIGURE 10.1. Simple model for enciphering bits

the plaintext to produce the ciphertext, namely combining the plaintext with a "random stream". The recipient can recreate the original plaintext by applying the inverse operation, in this case by combining the ciphertext with the same random stream.

This is particularly efficient since we can use the simplest operation available on a computer, namely exclusive-or $\oplus$. We saw in Chapter 9 that if the key is different for every message and the key is as long as the message, then such a system can be shown to be perfectly secure, namely we have the one-time pad. However, the one-time pad is not practical in many situations.

- We would like to use a short key to encrypt a long message.
- We would like to reuse keys.

Modern symmetric ciphers allow both of these possibilities, but thereby forfeit the perfect secrecy property. Such a trade-off is worthwhile because using a one-time pad produces horrendous key distribution problems. We shall see that key distribution is still problematic even for short, reusable keys.

There are a number of ways to attack a bulk cipher, some of which we outline below. We divide our discussion into passive and active attacks; a passive attack is generally easier to mount than an active attack.

- **Passive Attacks:** Here the adversary is only allowed to listen to encrypted messages. Then she attempts to break the cryptosystem by either recovering the key or determining some secret that the communicating parties did not want leaked. One common form of passive attack is that of traffic analysis, a technique borrowed from armies in World War

I, where a sudden increase in radio traffic at a certain point on the Western Front would signal an imminent offensive.

- **Active Attacks:** Here the adversary is allowed to insert, delete or replay messages between the two communicating parties. A general requirement is that an undetected insertion attack should require the breaking of the cipher, whilst the cipher needs to allow detection and recovery from deletion or replay attacks.

Bulk symmetric ciphers essentially come in two variants: stream ciphers, which operate on one data item (bit/letter) at a time, and block ciphers, which operate on data in blocks of items (e.g. 64 bits) at a time. In this chapter we look at historical stream ciphers, leaving modern stream ciphers until Chapter 12 and modern block ciphers until Chapter 13.

10.2. Stream Cipher Basics

Figure 10.2 gives a simple explanation of a stream cipher. This is very similar to our previous simple model, except the random bit stream is now produced from a short secret key using a public algorithm, called the keystream generator.

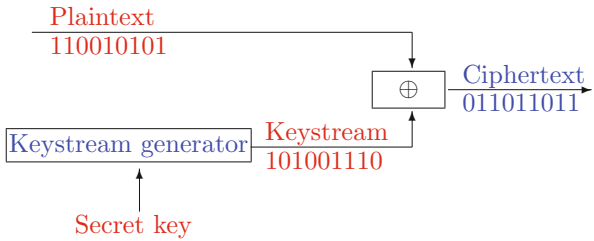


FIGURE 10.2. Stream ciphers

Thus we have $c_i = m_i \oplus k_i$ where

- $m_0, m_1, \ldots$ are the plaintext bits,
- $k_0, k_1, \ldots$ are the keystream bits,
- $c_0, c_1, \ldots$ are the ciphertext bits.

This means

$$m_i = c_i \oplus k_i$$

i.e. decryption is the same operation as encryption.

Stream ciphers such as that described above are simple and fast to implement. They allow very fast encryption of large amounts of data, so they are suited to real-time audio and video signals. In addition there is no error propagation; if a single bit of ciphertext gets mangled during transit (due to an attacker or a poor radio signal) then only one bit of the decrypted plaintext will be affected. They are very similar to the Vernam cipher mentioned earlier, except now the keystream is only pseudo-random as opposed to truly random. Thus whilst similar to the Vernam cipher they are *not* perfectly secure.

Just like the Vernam cipher, stream ciphers suffer from the following problem: the same key used twice gives the same keystream, which can reveal relationships between messages. For example suppose m_1 and m_2 were encrypted under the same key k, then an adversary could work out the exclusive-or of the two plaintexts without knowing what the plaintexts were

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2.$$

Hence, there is a need to change keys frequently either on a per message or on a per session basis. This results in difficult key management and distribution challenges, which, as we shall see later, can

be addressed using public key cryptography. A typical strategy is to use public key cryptography to determine session or message keys, and then to rapidly encrypt the actual data using either a stream or block cipher.

The keystream produced by the keystream generator above needs to satisfy a number of properties for the stream cipher to be considered secure. As a bare minimum it should

- Have a long period. Since the keystream k_i is produced via a deterministic process from the key, there will exist a number N such that
$$k_i = k_{i+N}$$
for all values of i. This number N is called the period of the sequence, and should be large for the keystream generator to be considered secure.
- Have pseudo-random properties. The generator should produce a sequence which appears to be random, in other words it should pass a number of statistical random number tests.
- Have large linear complexity (see Chapter 12 for an explanation).

However, these conditions are not sufficient. Generally, determining more of the sequence from a part should be computationally infeasible. Ideally, even if one knows the first one billion bits of the keystream sequence, the probability of guessing the next bit correctly should be no better than one half.

In Chapter 12 we shall discuss how modern stream ciphers can be created using a combination of simple circuits called Linear Feedback Shift Registers. But first we will look at an earlier construction using rotor machines, or in modern nomenclature Shift Registers (i.e. shift registers with no linear feedback).

10.3. The Lorenz Cipher

The Lorenz cipher was a German cipher from World War II which was used for strategic information, as opposed to the tactical and battlefield information encrypted under the Enigma machine. The Lorenz machine was a stream cipher which worked on streams of bits. However it produced not a single stream of bits, but five. The reason was due to the encoding of the teleprinter messages used at the time, namely Baudot code.

10.3.1. Baudot Code: To understand the Lorenz cipher we first need to understand Baudot code. We all are aware of the ASCII encoding for the standard letters on a keyboard, which uses seven bits for the data, plus one bit for error detection. Prior to ASCII, indeed as far back as 1870, Baudot invented an encoding which used five bits of data. This was further developed until, by the 1930s, it was the standard method of communicating via teleprinter. The data was encoded via a tape, consisting of a sequence of five rows of holes/non-holes.

Those of us of a certain age in the United Kingdom can remember the football scores being sent in on a Saturday evening by teleprinter, and those who are even older can maybe recall the ticker-tape parades in New York: the ticker-tape was the remains of messages in Baudot code that had been transmitted between teleprinters. Those who can remember early dial-up modems will recall that the speeds were measured in Baud, or characters per second, in memory of Baudot's invention.

Since five bits does not allow one to encode all the characters that one wants, Baudot code used two possible "states" called *letters shift* and *figures shift*. Movement between the two states was controlled by control characters; a number of other control characters were reserved for things such as space (SP), carriage return (CR), line feed (LF) or a character which rang the teleprinter's bell (BELL) (such control codes still exist in ASCII)[1]. The table for Baudot code in the 1930s is presented in Table 10.1. Thus to transmit the message

[1]A line feed moves one line down, whereas a carriage return moves the cursor to the beginning of a line. These two teleprinter/typewriter codes still cause problems today. In Windows, text files use both codes to move down and

Please, Please Help!

one would need to transmit the encoding, which we give in hexadecimal,

16, 12, 01, 03, 05, 01, 1B, 0C, 1F, 04, 16, 12, 01, 03, 05, 01, 04, 14, 01, 12, 16, 1B, 0D.

Bits in Code					Hex	Letters	Figures
lsb				msb	Code	Shift	Shift
0	0	0	0	0	00	NULL	NULL
1	0	0	0	0	01	E	3
0	1	0	0	0	02	LF	LF
1	1	0	0	0	03	A	-
0	0	1	0	0	04	SP	SP
1	0	1	0	0	05	S	’
0	1	1	0	0	06	I	8
1	1	1	0	0	07	U	7
0	0	0	1	0	08	CR	CR
1	0	0	1	0	09	D	ENQ
0	1	0	1	0	0A	R	4
1	1	0	1	0	0B	J	BELL
0	0	1	1	0	0C	N	,
1	0	1	1	0	0D	F	!
0	1	1	1	0	0E	C	:
1	1	1	1	0	0F	K	(
0	0	0	0	1	10	T	5
1	0	0	0	1	11	Z	+
0	1	0	0	1	12	L	)
1	1	0	0	1	13	W	2
0	0	1	0	1	14	H	£
1	0	1	0	1	15	Y	6
0	1	1	0	1	16	P	0
1	1	1	0	1	17	Q	1
0	0	0	1	1	18	O	9
1	0	0	1	1	19	B	?
0	1	0	1	1	1A	G	&
1	1	0	1	1	1B	Figures	Figures
0	0	1	1	1	1C	M	.
1	0	1	1	1	1D	X	/
0	1	1	1	1	1E	V	=
1	1	1	1	1	1F	Letters	Letters

TABLE 10.1. The Baudot code

start a new line of text, whereas Unix systems achieve the same effect by just using a line feed control code. This causes problems when a text file is moved from one system to another.

10.3.2. Lorenz Operation: The Lorenz cipher encrypted data in Baudot code form by producing a sequence of five random bits which was exclusive-or'd with the bits representing the Baudot code. The actual Lorenz cipher made use of a sequence of wheels, each having a number of pins. The presence, or absence, of a pin signalled a one or a zero signal. As the wheel turned, the position of the pins changed relative to an input signal. In modern parlance each wheel corresponds to a shift register.

Consider a register of length 32 bits or, equivalently, a wheel with circumference 32. At each clock tick the register shifts left by one bit and the leftmost bit is output; equivalently the wheel turns around 1/32 of a revolution and the topmost pin is taken as the output. This is represented in Figure 10.3. In Chapter 12 we shall see shift registers, with more complex feedback functions, being used in modern stream ciphers. However, it is interesting to see how similar ideas were used such a long time ago.

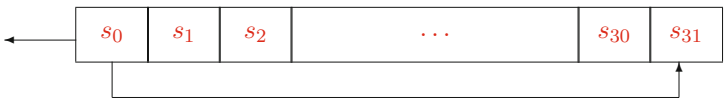


FIGURE 10.3. Shift Register of 32 bits

In Chapter 12 we shall see that the main problem is how to combine the more complex shift registers into a secure cipher. The same problem exists with the Lorenz cipher: namely, how the relatively simple operation of the wheels/shift registers can be combined to produce a cipher which is hard to break. From now on we shall refer to these as shift registers as opposed to wheels.

10.3.3. The Lorenz Cipher's Wheels: A Lorenz cipher uses twelve registers to produce the five streams of random bits. The twelve registers are divided into three subsets. The first set, called the chi-wheels, consists of five shift registers which we denote by $\chi_j^{(i)}$ comprising the output bit of the ith shift register on the jth clocking of the register, for $i = 1, 2, 3, 4, 5$. The five χ registers have lengths 41, 31, 29, 26 and 23, thus

$$\chi_{t+41}^{(1)} = \chi_t^{(1)},\ \chi_{t+31}^{(2)} = \chi_t^{(2)},\ \chi_{t+29}^{(3)} = \chi_t^{(3)},\ \chi_{t+26}^{(4)} = \chi_t^{(4)},\ \chi_{t+23}^{(5)} = \chi_t^{(5)}.$$

for all values of t. The second set of five shift registers, called the psi-wheels, we denote by $\psi_j^{(i)}$ for $i = 1, 2, 3, 4, 5$. These ψ registers have respective lengths 43, 47, 51, 53 and 59, i.e.

$$\psi_{t+43}^{(1)} = \psi_t^{(1)},\ \psi_{t+47}^{(2)} = \psi_t^{(2)},\ \psi_{t+51}^{(3)} = \psi_t^{(3)},\ \psi_{t+53}^{(4)} = \psi_t^{(4)},\ \psi_{t+59}^{(5)} = \psi_t^{(5)}.$$

The other two registers we shall denote by $\mu_j^{(i)}$ for $i = 1, 2$; these are called the motor registers. The lengths of the μ registers are 61 and 37 respectively, and so

$$\mu_{t+61}^{(1)} = \mu_t^{(1)},\ \mu_{t+37}^{(2)} = \mu_t^{(2)}.$$

10.3.4. Lorenz Cipher Operation: To describe how the Lorenz cipher clocks the various registers, we use the variable t to denote a global clock, which will be ticked for every Baudot code character which is encrypted. We also use a variable t_ψ to denote how often the ψ registers have been clocked, and a variable t_μ which denotes how often the second μ register, $\mu^{(2)}$, has been clocked. To start the cipher we set $t = t_\psi = t_\mu = 0$; at a given point we perform the following operations:

(1) Let κ denote the vector $(\chi_t^{(i)} \oplus \psi_{t_\psi}^{(i)})_{i=1}^5$.
(2) $t = t + 1$.

(3) If $\mu_t^{(1)} = 1$ then set $t_\mu = t_\mu + 1$.
(4) If $\mu_{t_\mu}^{(2)} = 1$ then set $t_\psi = t_\psi + 1$.
(5) Output κ.

The first line of the above produces the output keystream, the third line clocks the second μ register if the output of the first μ register is set (once it has been clocked), whilst the fourth line clocks all of the ψ registers if the output of the second μ register is set. From the above it should be deduced that the χ registers and the first μ register are clocked at every time interval. To encrypt a character the output vector κ is exclusive-or'd with the Baudot code representing the character of the plaintext. This is described graphically in Figure 10.4. Each clocking signal is depicted as a line with a circle on the end; each output wire is depicted by an arrow.

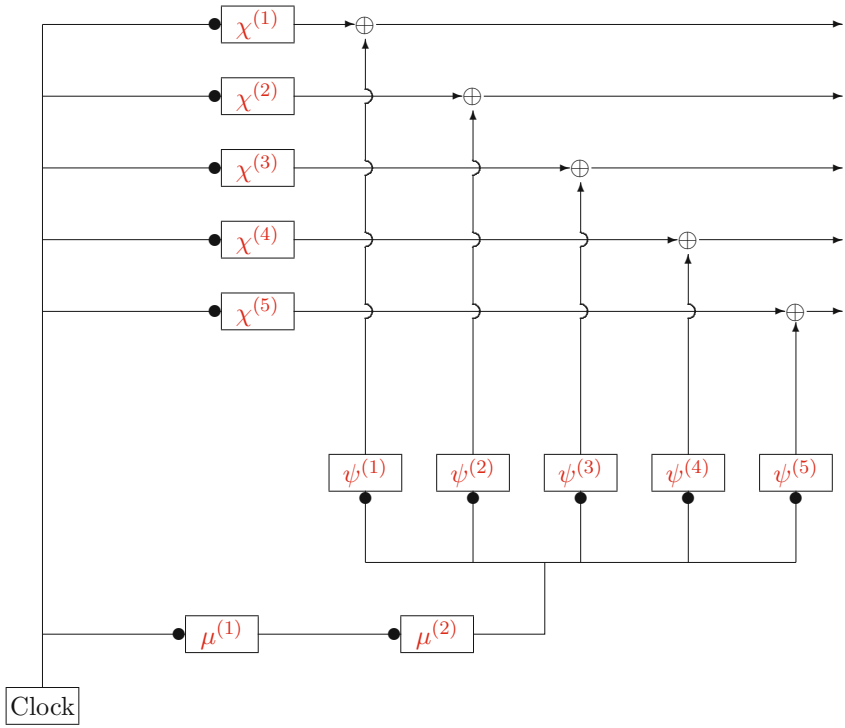


FIGURE 10.4. Graphical representation of the Lorenz cipher

The actual outputs of the ψ and μ motors at each time step are called the extended-ψ and the extended-μ streams. To ease future notation we will let $\psi'^{(i)}_t$ denote the output of the ψ registers at time t, whilst $\mu'^{(2)}_t$ will denote the output of the second μ register at time t. In other words, for a given tuple (t, t_ψ, t_μ) of valid clock values we have $\psi'^{(i)}_t = \psi^{(i)}_{t_\psi}$ and $\mu'^{(2)}_t = \mu^{(2)}_{t_\mu}$.

10.3.5. Example: To see this in operation consider the following example, where we describe the state of the cipher with the following notation:

```
Chi:    1111100010101100011110001011101000100011
        1100001101011101101011011001000
        1000100111100110001110111101O
        1111000110100010001110100l
```

```
          11011110000001010001110
Psi:      10110001101001010011010101010101101010101000
          11010101010101011101101010101011000101010101110
          10100001001101010101010001011010111010101010010100101001
          01010110101010000101010011011010100110110101101011001
          01010101010101010101101010010011010100100101010100100010010010
Motor:    01011111101101011001000111011110001001001110001111101011101000
          01110111000111111111000001010111111111
```

This gives the states of the χ, ψ and μ registers at time $t = t_\psi = t_\mu = 0$. The states will be shifted leftwise, and the output of each register will be the leftmost bit. So executing the above algorithm at time $t = 0$ the output first key vector will be

$$\kappa_0 = \chi_0 \oplus \psi_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}.$$

Since $\mu_1^{(1)} = 1$ we clock the t_μ value, and since $\mu_1^{(2)} = 1$ we also clock the t_ψ value. Thus at time $t = 1$ the state of the Lorenz cipher becomes

```
Chi:      11110001010110001111000101110100010001111
          10000110101110110101101100100001
          00010011110011000111011110101
          11100011010001000111010011
          10111100000010100011101
Psi:      01100011010010100110101010101101010101001
          10101010101010111011010101010110001010101011101
          01000010011010101010100001011010111010101010010011
          10101101010100000101010011011010100110110101101011010110010
          10101010101010101101010010011010100100101010100100010010100
Motor:    10111111011010110010001110111110001001001110001111101011101000
          11101110001111111111000010101111111110
```

Now we look at what happens at the next clock tick. At time $t = 1$ we now output the vector

$$\kappa_1 = \chi_1 \oplus \psi_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

But now since $\mu_{t+1}^{(1)}$ is equal to zero we do not clock t_μ, whilst since $\mu_1^{(2)} = 1$ we still clock the t_ψ value. This process is then repeated, so that we obtain the following sequence for the first 60 output values of the keystream κ_t:

```
010010000101001011101100011011011101110001111111000000001001
000100011101110011111010111110011000011011000111111101110111
001010010110011011101110100001000100111100110010101101010000
101000101101110010011011001011000110100011110001111101010111
100011001000010001001000000010100000010100011100001001101001 1
```

This is produced by exclusive-or'ing the output of the χ registers, which is given by:

$$\begin{array}{l}
11111000101011000111100010111010001000111111110001010110001 1\\
11000011010111011010110110010001100001101011101101011011 0010\\
10001001111001100011101111010100010011110011000111011110 1010\\
11110001101000100011101001111100011010001000111010011111 0001\\
11011110000001010001110110111100000010100011101101111 0000001
\end{array}$$

with the values of output of the ψ'_t stream at time t,

$$\begin{array}{l}
10110000111111101001010011010111111111111000001101010110 1010\\
11010010100000010101011101101000000000000111110010101100 0101\\
10100000100000001101010101010000000000000000000011011010 111010\\
01010011011111101010000101010000000000000111111101101010 0110\\
01010010100000001010101011010100000000000000000011001101 010010
\end{array}$$

To ease understanding we also present the output $\mu'^{(2)}_t$ which is

$$11110111100000111111111111111000000000001000010111111111111$$

Recall that a one in this stream means that the ψ registers are clocked whilst a zero implies they are not clocked. One can see this effect in the ψ'_t output given earlier.

10.3.6. Lorenz Key Size: Just like the Enigma machine the Lorenz cipher has a long-term key set-up and a short-term per message set-up. The long term key is the state of each register. Thus it appears that there are a total of

$$2^{41+31+29+26+23+43+47+51+53+59+61+37} = 2^{501}$$

states, although the actual number is slightly less than this due to a small constraint which will be introduced in a moment. In the early stages of the war the μ registers were changed on a daily basis, the χ registers were changed on a monthly basis and the ψ registers were changed on a monthly or quarterly basis. Thus, if the month's settings had been broken then the "day" key "only" consisted of at most

$$2^{61+37} = 2^{98}$$

states. As the war progressed the Germans moved to changing all the internal states of the registers every day.

Given these "day" values for the register contents, the per message setting is given by the starting position of each register. Thus the total number of message keys, given a day key, is given by

$$41 \cdot 31 \cdot 29 \cdot 26 \cdot 23 \cdot 43 \cdot 47 \cdot 51 \cdot 53 \cdot 59 \cdot 61 \cdot 37 \approx 2^{64}.$$

The Lorenz cipher has an obvious weakness as defined, which is what eventually led to its breaking, and which the Germans were aware of. The basic technique which we will use throughout the rest of this chapter is to take the "Delta" of a sequence – this is defined as follows, for a sequence $s = (s_i)_{i=0}^{\infty}$:

$$\Delta s = (s_i \oplus s_{i+1})_{i=0}^{\infty}.$$

We shall denote the value of the Δs sequence at time t by $(\Delta s)_t$. The Δ operator is very important in the analysis of the Lorenz cipher because

$$\kappa_t^{(i)} = \chi_t^{(i)} \oplus \psi_{t_\psi}^{(i)}$$

and

$$\kappa_{t+1}^{(i)} = \chi_{t+1}^{(i)} \oplus \left(\mu'^{(2)}_t \cdot \psi_{t_\psi+1}\right) \oplus \left((\mu'^{(2)}_t - 1) \cdot \psi_{t_\psi}\right),$$

so that

$$\begin{aligned}
(\Delta\kappa)_t &= (\chi_t \oplus \chi_{t+1}) \oplus \left(\mu'^{(2)}_t \cdot (\psi_{t_\psi} \oplus \psi_{t_\psi+1})\right)\\
&= (\Delta\chi)_t \oplus \left(\mu'^{(2)}_t \cdot (\Delta\psi)_{t_\psi}\right).
\end{aligned}$$

Now if $\Pr[\mu'^{(2)}_t = 1] = \Pr[(\Delta_\psi)_{t_\psi} = 1] = 1/2$, as we would have by choosing the register states uniformly at random, then with probability 3/4 the value of the $\Delta\kappa$ stream reveals the value of the $\Delta\chi$ stream, which enables the adversary to recover the state of the χ registers relatively easily. Thus the Germans imposed a restriction on the key values so that

$$\Pr[\mu'^{(2)}_t = 1] \cdot \Pr[(\Delta_\psi)_{t_\psi} = 1] \approx 1/2.$$

In what follows we shall denote these two probabilities by $\delta = \Pr[\mu'^{(2)}_t = 1]$ and $\epsilon = \Pr[(\Delta_\psi)_{t_\psi} = 1]$.

Finally, to fix notation, if we let the Baudot encoding of the message be given by the sequence ϕ of 5-bit vectors, and the ciphertext be given by the sequence γ, then we have

$$\gamma_t = \phi_t \oplus \kappa_t.$$

As the war progressed more complex internal operations of the Lorenz cipher were introduced. These were called "limitations" by Bletchley, and they introduced extra complications into the clocking of the various registers. We shall ignore these extra complications however in our discussion.

Initially the Allies did not know anything about the Lorenz cipher, even that it consisted of twelve wheels, let alone their period. In August 1941 the Germans made a serious mistake: they transmitted almost identical messages, of roughly 4 000 characters in length, using exactly the same key. From this the cryptanalyst John Tiltman managed to reconstruct the key of roughly 4 000 characters that had been output by the Lorenz cipher. From this sequence of apparently random strings of five bits another cryptographer, Bill Tutte, recovered the precise internal workings of the Lorenz cipher. The final confirmation that the internal workings had been deduced correctly did not come until the end of the war, when the Allies captured a Lorenz machine on entering Germany.

10.4. Breaking the Lorenz Cipher's Wheels

Having determined the structure of the Lorenz cipher the problem remained of how to break it. The attack method used was broken into two stages. In the first stage the wheels needed to be broken: this was an involved process which only had to be performed once for each wheel configuration. Then a simpler procedure was produced which recovered the wheel positions for each message.

We now explain how wheel breaking occurred. The first task was to obtain with reasonable certainty the value of the sequence

$$\Delta\kappa^{(i)} \oplus \Delta\kappa^{(j)}$$

for different distinct values of i and j, usually $i = 1$ and $j = 2$. There were various different ways of performing this; below we present a gross simplification of the techniques used by the cryptanalysts at Bletchley. Our goal is simply to show that breaking even a 60 year old stream cipher requires some intricate manipulation of probability estimates, and that even small deviations from randomness in the output stream can cause a catastrophic failure in security.

To do this we first need to consider some characteristics of the plain text. Standard natural language contains a larger sequence of repeated characters than one would normally expect, compared to the case when a message is just random gibberish. If messages were random then one would expect

$$\Pr[(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0] = 1/2,$$

for any $i, j \in \{1, 2, 3, 4, 5\}$. However, if the plaintext sequence contains slightly more repeated characters than we expect, this probability would be slightly more than 1/2, so we set

$$\Pr[(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0] = 1/2 + \rho. \tag{11}$$

Due to the nature of German military parlance, and the Baudot encoding method, this was apparently particularly pronounced when one considered the first and second streams of bits, i.e. $i = 1$ and $j = 2$.

There are essentially two situations for wheel breaking. The first (more complex) case is when we do not know the underlying plaintext for a message, i.e. the attacker only has access to the ciphertext. The second case is when the attacker can guess with reasonable certainty the value of the underlying plaintext (a "crib" in the Bletchley jargon), and so can obtain the resulting keystream.

10.4.1. Ciphertext Only Method: The basic idea is that the sequence of ciphertext Deltas,

$$\Delta\gamma^{(i)} \oplus \Delta\gamma^{(j)}$$

will "reveal" the true value of the sequence

$$\Delta\chi^{(i)} \oplus \Delta\chi^{(j)}.$$

Consider the probability that we have

$$(\Delta\gamma^{(i)})_t \oplus (\Delta\gamma^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t. \tag{12}$$

Because of the relationship

$$\begin{aligned}(\Delta\gamma^{(i)})_t \oplus (\Delta\gamma^{(j)})_t &= (\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t \oplus (\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t \\ &= (\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t \oplus (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \\ &\quad \oplus \left(\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right)\right),\end{aligned}$$

equation (12) can hold in one of two ways:

- Either we have
$$(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 0$$
and
$$\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right) = 0.$$
The first of these events occurs with probability $1/2+\rho$ by equation (11), whilst the second occurs with probability
$$(1-\delta) + \delta \cdot (\epsilon^2 + (1-\epsilon)^2) = 1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta.$$
- Or we have
$$(\Delta\phi^{(i)})_t \oplus (\Delta\phi^{(j)})_t = 1$$
and
$$\mu'^{(2)}_t \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right) = 1.$$
The first of these events occurs with probability $1/2-\rho$ by equation (11), whilst the second occurs with probability
$$2 \cdot \delta \cdot \epsilon \cdot (1-\epsilon).$$

Combining these probabilities together we find that equation (12) holds with probability

$$\begin{aligned}&(1/2+\rho) \cdot (1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta) + 2 \cdot (1/2-\rho) \cdot \delta \cdot \epsilon \cdot (1-\epsilon) \\ &\quad \approx (1/2+\rho) \cdot \epsilon + (1/2-\rho) \cdot (1-\epsilon) \\ &\quad = 1/2 + \rho \cdot (2 \cdot \epsilon - 1),\end{aligned}$$

since $\delta \cdot \epsilon \approx 1/2$ due to the key generation method mentioned earlier. So assuming we have a sequence of n ciphertext characters, if we are trying to determine

$$\sigma_t = (\Delta\chi^{(1)})_t \oplus (\Delta\chi^{(2)})_t,$$

i.e. we have set $i = 1$ and $j = 2$, then we know that this latter sequence has period $1271 = 41 \cdot 31$. Thus each element in this sequence will occur $n/1271$ times. If n is large enough, then taking a majority verdict will determine the value of the sequence σ_t with some certainty.

10.4.2. Known Keystream Method: Now assume that we know the value of $\kappa_t^{(i)}$. We use a similar idea to above, but now we use the sequence of keystream Deltas,

$$\Delta\kappa^{(i)} \oplus \Delta\kappa^{(j)}$$

and hope that this reveals the true value of the sequence

$$\Delta\chi^{(i)} \oplus \Delta\chi^{(j)}.$$

This is likely to happen due to the identity

$$(\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \oplus \left(\mu_t'^{(2)} \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right)\right).$$

Hence we will have

$$(\Delta\kappa^{(i)})_t \oplus (\Delta\kappa^{(j)})_t = (\Delta\chi^{(i)})_t \oplus (\Delta\chi^{(j)})_t \tag{13}$$

precisely when

$$\mu_t'^{(2)} \cdot \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right) = 0.$$

This last equation will hold with probability

$$\begin{aligned}(1-\delta) + \delta \cdot (\epsilon^2 + (1-\epsilon)^2) &= 1 - 2 \cdot \epsilon \cdot \delta + 2 \cdot \epsilon^2 \cdot \delta \\ &\approx 1 - 1 + \epsilon = \epsilon,\end{aligned}$$

since $\delta \cdot \epsilon \approx 1/2$. But since $\delta \cdot \epsilon \approx 1/2$ we usually have $0.6 \leq \epsilon \leq 0.8$, thus equation (13) holds with a reasonable probability. So as before we try to take a majority verdict to obtain an estimate for each of the 1271 terms of the σ_t sequence.

10.4.3. Both Methods Continued: Whichever of the above methods we use there will still be some errors in our guess for the stream σ_t, which we will now try to correct. In some sense we are not really after the values for the sequence σ_t, what we really want is the exact values of the two shorter sequences

$$(\Delta\chi^{(1)})_t \text{ and } (\Delta\chi^{(2)})_t,$$

since these will allow us to deduce possible values for the first two χ registers in the Lorenz cipher. The approximation for the σ_t sequence of 1271 bits we now write down in a 41×31-bit array, with the first 31 bits in row one, the second 31 bits in row two, and so on. A blank is placed in the array if we cannot determine the value of this bit with any reasonable certainty. For example, assuming the above configuration was used to encrypt the ciphertext, we could obtain an array which looks something like this:

```
0-0---01--1--110--110-10--1-0-1
010---0---10011-1----1-010-1--1
--00-1--111001--1-1--1101--1001
0-00--01----0-------0-10-0--001
-0-110-000-110-1000------1---10
-1--0---1-100110111-01---01---1
01-0-10111-001101-------1-1-0--
-01--0-0--0-100-00001-0---0----
1--1--10000-------0------1--11-
101---1-00-110--0-00--0-0---100
1---11--000---0---0-1--1------1
---1-0----011--1----1---01-01-0
-1---1--1--00110-1-1-110-0-100-
1-------10--10-1---0100-010--1-
0--0--011--0011--11-011------0-
--0001-1-11--11011---1-010110--
```

```
----10---------1000--001-10----
0-00-1-11110----1111-1-01-11-0-
----01-1-----11--111011-1--10--
0-0-01--1---011---11--1-1--1001
10-1---0-1-1-0010--01-0--10--10
----------1-01-0-1--0-10--1-001
010-01-1-110011-11---1-0---1--1
1-1-1-1000-11-010--01-0101-01-0
1---10-00--110---0-0--011-00-10
10-1-010-0----01----1--10-0----
-1--0-011-1001-0----01101011--1
010-010-11-0--101--10--0--1--0-
-01---1-0---1-01000-1---0-001-0
--1-10--0--110-100001-0--10-110
----1--00001-00--00010010----10
011-0101-110-1-011-101101----01
01---1----100--01---01-01--0-01
-011--1-000-1--10-00-0---1001-0
1011-01--0---001---01-01--0----
---0--0-1-1--11----1----1-11---
----0---1--0---0-----11--0--00-
10--101--0----0-0-0--0--010----
-100---1-110-----11-01-0---1---
0100----1-1----01-1--1111--11--
0-0001-1-1-0011011-1---01011-0-
```

Now the goal of the attacker is to fill in this array, a process known at Bletchley as "rectangling", bearing in mind that some of the zeros and ones entered could themselves be incorrect. The point to note is that, when completed, there should be just two distinct rows in the table, each the complement of the other, and similarly for the columns. A reasonable method to follow is to take all rows starting with zero and then count the number of zeros and ones in the second element of those rows. In our example above, we find there are seven ones and no zeros. Doing the same for rows starting with a one we find there are four zeros and no ones. Thus we can deduce that the two types of rows in the table should start with a `10` and a `01`. We then fill in the second element in any row which has its first element set. We continue in this way, first looking at rows and then looking at columns, until the whole table is filled in.

The above table was found using a few thousand characters of known keystream, which allows (via the above method) the simple reconstruction of the full table. According to the Bletchley documents, the cryptographers at Bletchley would actually use a few hundred characters of keystream in a known keystream attack, and a few thousand in an unknown keystream attack. Since we are following rather naive methods our results are not as spectacular.

Once completed we can take the first column as the value of the $(\Delta\chi^{(1)})_t$ sequence and the first row as the value of the $(\Delta\chi^{(2)})_t$ sequence. We can then repeat this analysis for different pairs of the χ registers until we determine that we have

$$\begin{aligned}
\Delta\chi^{(1)} &= 0000100111110100100010011100111001001000,\\
\Delta\chi^{(2)} &= 0100010111100110111101101011001,\\
\Delta\chi^{(3)} &= 1001101000101010010011000111 1,\\
\Delta\chi^{(4)} &= 0001001011100110010011101 0,\\
\Delta\chi^{(5)} &= 01100010000011110010011.
\end{aligned}$$

From these $\Delta\chi$ sequences we can then determine possible values for the internal state of the χ registers.

10.4.4. Breaking the Other Wheels: So having "broken" the χ wheels of the Lorenz cipher, the task remains to determine the internal state of the other registers. In the ciphertext only attack one now needs to recover the actual keystream, a step which is clearly not needed in the known-keystream scenario. The trick here is to use the statistics of the underlying language again to try to recover the actual $\kappa^{(i)}$ sequence. We first de-χ the ciphertext sequence γ, using the values of the χ registers which we have just determined, to obtain

$$\begin{aligned}\beta_t^{(i)} &= \gamma_t^{(i)} \oplus \chi_t^{(i)} \\ &= \phi_t^{(i)} \oplus {\psi'}_t^{(i)}.\end{aligned}$$

We then take the Delta of this β sequence

$$(\Delta\beta)_t = (\Delta\phi)_t \oplus \left(\mu'^{(2)}_t \cdot (\Delta\psi)_{t_\psi}\right),$$

and by our previous argument we will see that many values of the $\Delta\phi$ sequence will be "exposed" in the $\Delta\beta$ sequence. Using a priori knowledge of the $\Delta\phi$ sequence, for example that it uses Baudot codes and that natural language has many sequences of bigrams (e.g. a space always follows a full stop), one can eventually recover the sequence ϕ and hence κ. At Bletchley this last step was usually performed by hand.

So in both scenarios we have now determined both the χ and the κ sequences. But what we are really after is the initial values of the registers ψ and μ. To determine these we de-χ the resulting κ sequence to obtain the ${\psi'}_t$ sequence. In our example this would reveal the sequence

```
10110000111111010010100110101111111111100000110101011010 10
11010010100000010101011101101000000000001111100101011000101
10100000100000001101010101010000000000000000011011010111010
01010011011111101010000101010000000000000111111011010100110
01010010100000010101010110101000000000000000011001101010010
```

given earlier. From this we can then recover a guess as to the $\mu'^{(2)}_t$ sequence.

```
11110111100000111111111111111000000000010000101111111111111...
```

Note that it is only a guess; it might occur that $\psi_{t_\psi} = \psi_{t_\psi+1}$, but we shall ignore this possibility. Once we have determined enough of the $\mu'^{(2)}_t$ sequence so that we have 59 ones in it, then we will have determined the initial state of the ψ registers. This is because after 59 clock ticks of the ψ registers all outputs have been presented in the ψ' sequence, since the largest ψ register has size 59.

All that remains is to determine the state of the μ registers. To do this we notice that the $\mu'^{(2)}_t$ sequence will make a transition from a 0 to a 1, or a 1 to a 0, precisely when $\mu_t^{(1)}$ outputs a one. By constructing enough of the $\mu'^{(2)}_t$ stream as above (say a few hundred bits) this allows us to determine the value of the $\mu^{(1)}$ register almost exactly. Having recovered $\mu_t^{(1)}$ we can then deduce the values which must be contained in $\mu_{t_\mu}^{(2)}$ from this sequence and the resulting value of $\mu'^{(2)}_t$.

According to various documents, in the early stages of the Lorenz cipher-breaking effort at Bletchley, the entire "Wheel Breaking" operation was performed by hand. However, as time progressed the part which involved determining the $\Delta\chi$ sequences above from the rectangling procedure was eventually performed by the Colossus computer.

10.5. Breaking a Lorenz Cipher Message

The Colossus was the world's first programmable digital electronic computer, and as such was the precursor of all modern computers. The role of the Colossus was vital to the Allied war effort, and

was so secret that its very existence was not divulged until the 1980s. The Colossus computer was originally created not to break the wheels, i.e. to determine the long-term key of the Lorenz cipher, but to determine the per message settings, and hence to help break the individual ciphertexts. Whilst the previous method for breaking the wheels could be used to attack any ciphertext, for it to work efficiently requires a large ciphertext and a lot of luck. However, once the wheels are broken, i.e. we know the bits in the various registers, breaking the next ciphertext becomes easier.

To break a message we again use the trick of de-χ'ing the ciphertext sequence γ, and then applying the Delta method to the resulting sequence β. We assume we know the internal states of all the registers but not their starting positions. We shall let s_i denote the unknown values of the starting positions of the five χ wheels and s_ϕ (resp. s_μ) the global unknown starting position of the set of ϕ (resp. μ) wheels.

$$\begin{aligned}\beta_t &= \gamma_t \oplus \chi_{t+s_p} \\ &= \phi_t \oplus {\psi'}_{t+s_\phi},\end{aligned}$$

and then

$$(\Delta\beta)_t = (\Delta\phi)_{t+s_\phi} \oplus \left({\mu'}^{(2)}_{t+s_\mu} \cdot (\Delta\psi)_{t_\psi}\right).$$

We then take two of the resulting five bit streams and exclusive-or them together as before to obtain

$$\begin{aligned}(\alpha^{(i,j)})_t &= (\Delta\beta^{(i)})_t \oplus (\Delta\beta^{(j)})_t \\ &= (\Delta\phi^{(i)})_{t+s_\phi} \oplus (\Delta\phi^{(j)})_{t+s_\phi} \oplus {\mu'}^{(2)}_{t+t_\mu} \left((\Delta\psi^{(i)})_{t_\psi} \oplus (\Delta\psi^{(j)})_{t_\psi}\right).\end{aligned}$$

Using our prior probability estimates we can determine the following probability estimate

$$\Pr[(\alpha^{(i,j)})_t = 0] \approx 1/2 + \rho \cdot (2 \cdot \epsilon - 1),$$

which is exactly the same probability we had for equation (11) to hold true. In particular we note that $\Pr[{\alpha^{(i,j)}}_t = 0] > 1/2$, which forms the basis of this method of breaking into Lorenz ciphertexts.

Let us fix $i = 1$ and $j = 2$. On assuming we know the values for the registers, all we need do is determine their starting positions s_1, s_2. We simply need to go through all $1271 = 41 \cdot 31$ possible starting positions for the first and second χ registers. For each one of these starting positions we compute the associated $(\alpha^{(1,2)})_t$ sequence and count the number of values which are zero. Since we have $\Pr[\alpha_t^{(i,j)} = 0] > 1/2$ the correct value for the starting positions will correspond to a particularly high value for the count of the number of zeros.

This is a simple statistical test which allows one to determine the start positions of the first and second χ registers. Repeating this for other pairs of registers, or using similar statistical techniques, we can recover the start position of all χ registers. These statistical techniques are what the Colossus computer was designed to perform.

Once the χ register positions have been determined, the determination of the start positions of the ψ and μ registers can then be performed by hand. The techniques for this are very similar to the earlier techniques needed to break the wheels, however once again various simplifications occur since one is assumed to know the state of each register, but not its start position.

Chapter Summary

- We have described the general model for symmetric ciphers, and for stream ciphers in particular.

- We have looked at the Lorenz cipher as a stream cipher, and described its inner workings in terms of shift registers.
- We sketched how the Lorenz cipher was eventually broken, in particular how very tiny deviations from true randomness in the output were exploited by the Blecthley cryptographers.

Further Reading

The paper by Carter provides a more detailed description of the cryptanalysis performed at Bletchley on the Lorenz cipher. The book by Gannon is a very readable account of the entire operation related to the Lorenz cipher, from obtaining the signals through to the construction and operation of the Colossus computer. For the "real" details you should consult the General Report on Tunny.

F.L. Carter. *The Breaking of the Lorenz Cipher: An Introduction to the Theory Behind the Operational Role of "Colossus" at BP.* In Cryptography and Coding – 1997, LNCS 1355, 74–88, Springer, 1997.

P. Gannon *Colossus: Bletchley Park's Greatest Secret.* Atlantic Books, 2007.

J. Good, D. Michie and G. Timms. *General report on Tunny, With Emphasis on Statistical Methods.* Document reference HW 25/4 and HW 25/5, Public Record Office, Kew. Originally written in 1945, declassified in 2000.

Part 3

Modern Cryptography Basics

In this part we cover the basic components of modern cryptographic systems. As an overview of the chapter headings will show, modern cryptography is not just about symmetric encryption. We have other symmetric primitives such as message authentication codes, there are public key primitives such as public key encryption and digital signatures, and there are keyless primitives such as hash functions.

We also will see that behind each of these primitives is a notion of what it means for the primitive to be secure. This is the main distinction between cryptography in the twenty-first century and that which preceded it. Modern cryptography is as much about defining what we mean by something being secure as it is about actually coming up with something that achieves that security goal.

CHAPTER 11

Defining Security

Chapter Goals

- To explain the notion of a secure pseudo-random function and permutation.
- To explain the various notions of security of encryption schemes, especially the notion of indistinguishability of encryptions.
- To explain the various attack notions, in particular adaptive chosen ciphertext attacks.
- To show how the concept of non-malleability and adaptive chosen ciphertext attacks are related.
- To explain notions related to the security of signature schemes, message authentication codes and other cryptographic mechanisms.
- The chapter also introduces some basic techniques used in "security proofs".

11.1. Introduction

Modern cryptography is focused on three key aspects: definitions, schemes and proofs.

- **Definitions:** The first challenge modern cryptography addresses is to actually arrive at a concrete mathematical definition of what it means for a particular cryptographic mechanism to be secure. Whilst we may have a conceptual notion that encryption should be secure as long as the key is not revealed, it is not straightforward to define this precisely. We will also see that modern cryptography is about more than just encryption.
- **Schemes:** Once we have a security definition for a specific cryptographic mechanism, we need to design schemes which it is hoped will meet the security definition we have previous defined. For example we might want to build an encryption scheme whose security intuitively rests on the difficulty of factoring numbers.
- **Proofs:** The natural question to ask then is whether the design meets the security definition. This is the approach of "provable security", or more accurately "reductionist security". In this approach we design a scheme based on certain building blocks; for example a building block could be the difficulty of factoring large integers, or the security of a specific block cipher. Then we try to show that the larger scheme meets the security definition by showing that if it did not one could also "break" the simpler component (e.g. factor numbers).

This chapter is devoted to defining security, which in this book will be done in the same way as we looked at the factoring-related and discrete-logarithm-related problems in Chapters 2 and 3. In later chapters we will look at how to construct the mechanisms described in this chapter from various building blocks.

11.2. Pseudo-random Functions and Permutations

The security games from Chapters 2 and 3 for the factoring-related problems and the discrete-logarithm-related problems will form the cornerstone of what we will call public key cryptography.

For symmetric key cryptography we need to look at another basic primitive called a pseudo-random function, or PRF for short. A PRF is a function which appears to be random to the adversary; in other words the adversary cannot predict its output.

It might be tempting to first try to define a security game as in Figure 11.1, for a function F with domain D and codomain C. In this game we give the adversary the function F, and then pick either a random pair (x, y), if $b = 0$, or a pair depending on the function F, $(x, F(x))$, if $b = 1$. The adversary's goal is to guess whether she was given a real pair associated with the PRF function, or a random pair. However, this game is easy to win since the function F is known to the adversary. She can determine the bit b by evaluating F on x and checking whether it equals y. If so, then the bit is highly likely to be one, otherwise the bit is zero. But we want to be able to give the function to the adversary, so that she can inspect the code etc. So we seem to be stuck if we give the adversary the function.

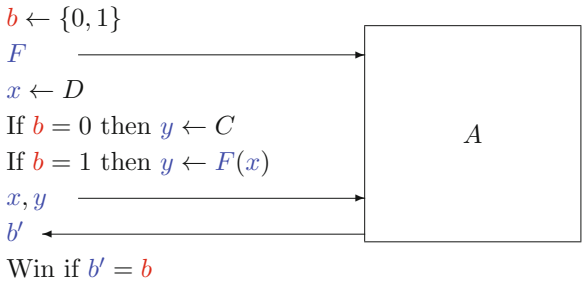


FIGURE 11.1. First attempt at a security game for a PRF

The way we solve this conundrum is by looking at Kerckhoffs' principle again. Recall that this decrees that the security of the public algorithm should rest entirely on the secrecy of the key. So instead of looking at a single function F we define a *family* of pseudo-random functions which are indexed by a key k chosen from some set K. This is much like what we were trying to achieve using the Lorenz stream cipher in Chapter 10, only where (unlike in World War II) the adversary starts by knowing the function family but not the key, and each key defines a new random function which should appear to the adversary.

This allows us to define the security of a pseudo-random function family $\{F_k\}_K$ according to the game given in Figure 11.2. We assume that the adversary is given a description of the function family, but not the specific function selected from the family. In our pictures the family is denoted by $\{F_k\}_K$, whereas the actual function selected is denoted by F_k. We also assume that each function has the same domain D and codomain C, and that the index k is chosen from a set K. The adversary's goal is to determine whether the function input/output pair she receives is from the real PRF-family, $\{F_k\}_K$, or from a random function. Note, we do not give the key k to the adversary.

However, this game gives far too much power to the challenger. It is highly unlikely that an adversary will be able to win this game except against very simple PRF families. And there are function families, for example taking $K = D = C = \{0,1\}^n$ and $F_k(x) = k \oplus x$, for which the game is unwinnable for even infinitely powerful adversaries.

To give the adversary a fighting chance, we would like to give her the ability to ask multiple queries on inputs x of her choosing. This will balance up the abilities of the adversary and the challenger, and hence make a more meaningful security definition for future use. So a better definition is given by the game in Figure 11.3; note that here the adversary can now ask various queries x of her choosing. We denote this in our diagrams by saying the adversary has access to an oracle $\mathcal{O}_{F_k}$ which she can call multiple times, and we place the "code" for such oracles on the

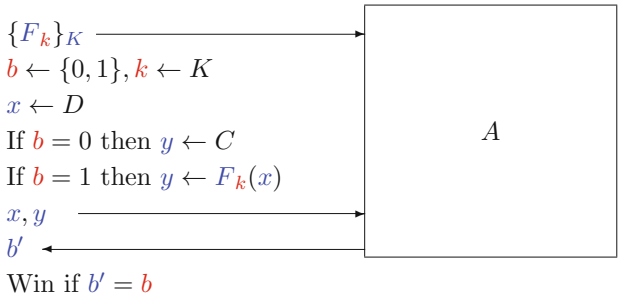


FIGURE 11.2. Second attempt at a security game for a PRF

right of the adversary. These "oracles" are subroutines that we give to the adversary; they provide access to functions on data which can be adaptively chosen by the adversary. However, in our PRF game we have to be a little careful to avoid giving different random answers in the case of $b = 0$ when the adversary asks the same x twice. Thus to avoid two different answers being given for the same query x, we record the answers provided via means of the state $\mathcal{L}$.

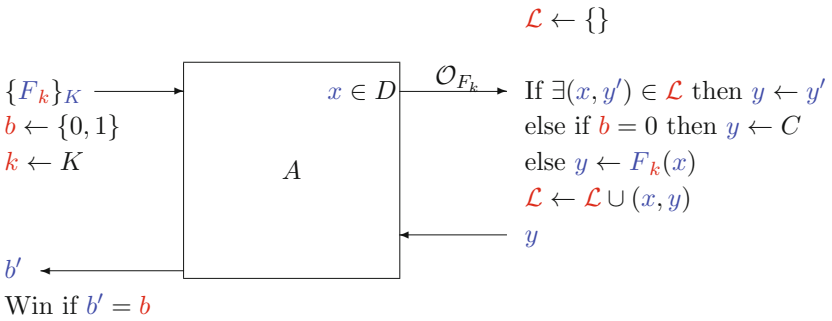


FIGURE 11.3. The final security game for a PRF

We define the advantage, much like we did for the Decision Diffie–Hellman problem, as follows,

$$\mathrm{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) = 2 \cdot \left| \Pr[A^{\mathcal{O}_{F_k}} \text{ wins}] - \frac{1}{2} \right|,$$

where we use the superscript $\mathcal{O}_{F_k}$ to indicate that the adversary has access to an oracle which computes the function in the forwards direction. Each call to the $\mathcal{O}_{F_k}$ oracle is counted as one time step, and so the number of oracle calls is bounded above by the running time of the adversary. If we want to make explicit the number of oracle calls we denote it by $q_{\mathcal{O}_{F_k}}$ and write the advantage as $\mathrm{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A; q_{\mathcal{O}_{F_k}})$.

Recall that as this is a decision game we need to subtract $1/2$ from the probability of the adversary winning, since she could just guess the bit b and still win with probability $1/2$. Thus by subtracting $1/2$ we ensure the advantage measures the extra power the adversary has over random guessing. In addition, we have the following analogue of Lemma 2.3,

Lemma 11.1. *Let A be an adversary against the* PRF *security of the function family $\{F_k\}_K$, then if b' is the bit chosen by A and b is the bit chosen by the challenger in the game, we have*

$$\mathrm{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) = \left| \Pr[b' = 1 \mid b = 1] - \Pr[b' = 1 \mid b = 0] \right|.$$

An important concept related to a PRF is that of a pseudo-random permutation (PRP). In this concept the domain and codomain are the same set D, and the function is one-to-one. Since an adversary can tell a PRF from a PRP if she discovers that the function is not one-to-one, we define the PRP security game as in Figure 11.4.

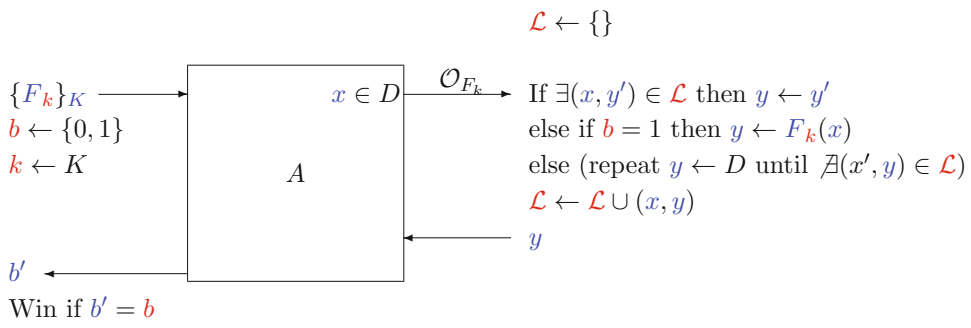

$\mathcal{L} \leftarrow \{\}$
$\{F_k\}_K$
$b \leftarrow \{0,1\}$
$k \leftarrow K$
$x \in D$
$\mathcal{O}_{F_k}$
If $\exists(x, y') \in \mathcal{L}$ then $y \leftarrow y'$
else if $b = 1$ then $y \leftarrow F_k(x)$
else (repeat $y \leftarrow D$ until $\nexists(x', y) \in \mathcal{L}$)
$\mathcal{L} \leftarrow \mathcal{L} \cup (x, y)$
y
A
b'
Win if $b' = b$

FIGURE 11.4. The security game for a PRP

Just as above we define the advantage in the obvious way, using the following notation:

$$\text{Adv}^{\text{PRP}}_{\{F_k\}_K}(A) = 2 \cdot \left| \Pr[A^{\mathcal{O}_{F_k}} \text{ wins}] - \frac{1}{2} \right|.$$

where we use the superscript $\mathcal{O}_{F_k}$ to indicate that the adversary has access to an oracle which computes the permutation in the forwards direction. We can also define a game in which the adversary gets access to an additional oracle which enables her to invert the permutation on elements of her choice as well. We denote the advantage then by the notation

$$\text{Adv}^{\text{PRP}}_{\{F_k\}_K}(A) = 2 \cdot \left| \Pr[A^{\mathcal{O}_{F_k}, \mathcal{O}_{F_k^{-1}}} \text{ wins}] - \frac{1}{2} \right|.$$

As an exercise you should draw a picture, as above, to describe the game that the adversary plays, and define the consistency checks that the challenger needs to perform in the case of $b = 0$.

Notice how we defined PRF security with the adversary unable to tell the difference between accessing the PRF and accessing a random function, and we defined PRP security with the adversary unable to tell the difference between accessing the PRP and accessing a random permutation. In both games the adversary has essentially the same interface to both the game and its oracle. An interesting question which immediately comes to mind is whether an adversary can tell the difference between a PRP family and a PRF family, when the PRP and PRF family have the same domain and codomain. We can think of taking *the same* adversary A and placing her in the PRF game or the PRP game; we can then ask, can she tell the difference?

To answer this question we need to define what we mean by "tell the difference". For this we mean that the adversary behaves differently; but the only thing the adversary does is output a bit b'. Thus we can think of telling the difference as meaning that there will be some difference in the advantage of the adversary A in one game compared to the other. Thus we want to bound

$$\left| \text{Adv}^{\text{PRF}}_{\{F_k\}_K}(A; q) - \text{Adv}^{\text{PRP}}_{\{F_k\}_K}(A; q) \right|$$

for some PRP family $\{F_k\}_K$ with domain/codomain D.

Lemma 11.2 (PRP-PRF Switching Lemma)**.** *Let A be an adversary and $\{F_k\}_K$ be a family of pseudo-random permutations with domain and codomain equal to D, then*

$$\left| \text{Adv}^{\text{PRF}}_{\{F_k\}_K}(A; q) - \text{Adv}^{\text{PRP}}_{\{F_k\}_K}(A, q) \right| < \frac{q^2}{|D|}.$$

PROOF. Suppose we run A in the PRF game, and we let E denote the event that the oracle called by A returns the same value in the codomain for two distinct input values. We have

$$\begin{aligned}\Pr[A \text{ wins the PRF game}] &= \Pr[A \text{ wins the PRF game} \mid E] \cdot \Pr[E] \\ &\quad + \Pr[A \text{ wins the PRF game} \mid \neg E] \cdot \Pr[\neg E] \\ &\leq \Pr[E] + \Pr[A \text{ wins the PRF game} \mid \neg E] \\ &= \Pr[E] + \Pr[A \text{ wins the PRP game} \mid \neg E].\end{aligned}$$

Now we note that the probability of E occurring is, from the birthday bound, at most $q^2/(2 \cdot |D|)$. So we have, where we assume the probability of A winning is always at least $1/2$ (which we can do without loss of generality),

$$\begin{aligned}&\left|\mathrm{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A;q) - \mathrm{Adv}^{\mathsf{PRP}}_{\{F_k\}_K}(A;q)\right| \\ &\quad = \left|2 \cdot \left|\Pr[A \text{ wins the PRF game}] - \frac{1}{2}\right|\right. \\ &\qquad \left. -2 \cdot \left|\Pr[A \text{ wins the PRP game}] - \frac{1}{2}\right|\right| \\ &\quad = \Big|2 \cdot \Pr[A \text{ wins the PRF game}] - 1 \\ &\qquad - 2 \cdot \Pr[A \text{ wins the PRP game}] + 1\Big| \\ &\quad \leq 2 \cdot \Pr[E] \qquad \text{by above} \\ &\quad \leq \frac{q^2}{|D|} \qquad \text{by the birthday bound.}\end{aligned}$$

□

11.3. One-Way Functions and Trapdoor One-Way Functions

The discrete logarithm problem is an example of a one-way function: we give the adversary a public function (in this case the function $f(x) = g^x$) and ask her to invert the function on an element of the challenger's choosing. In terms of pictures this is given by Figure 11.5, for a function F with domain D and codomain C. Notice the similarity between this diagram and Figure 3.1. Note that we do not need to give the adversary an oracle to query values of F of her choosing since the function F is given to the adversary. We define the advantage $\mathrm{Adv}^{\mathsf{OWF}}_F(A)$ in the usual way as

$$\mathrm{Adv}^{\mathsf{OWF}}_F(A) = \Pr[A \text{ wins the OWF game}].$$

Notice how we can recast the discrete logarithm problem as an example of a one-way function. Thus the DLP security game is the same as the OWF game for the specific function $F : x \longrightarrow g^x$ in the group generated by g.

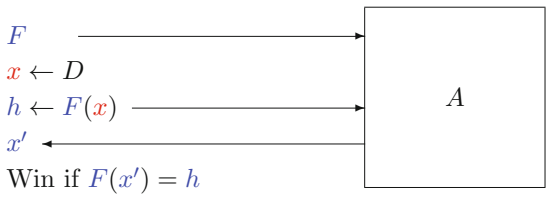


FIGURE 11.5. Security game for a one-way function

Now look again at another of our number-theoretic functions: the RSA problem as in Figure 2.2. Recall that this was the problem, given e and N such that $\gcd(e,(p-1)(q-1))=1$ and a value y, to find x such that

$$x^e = y \pmod{N}.$$

This is similar to the problem of inverting the one-way function given by $F_{N,e}(x) = x^e \pmod{N}$; however it is more than that. The function $F_{N,e}(x)$ in the RSA problem has an extra property: there exists a value d which allows one to *efficiently* invert the function. The value d is called the *trapdoor*, and such functions are called trapdoor one-way functions. In fact, even more is true: since the RSA function $F_{N,e}(x) = x^e \pmod{N}$ acts as a permutation on the group $\mathbb{Z}/N\mathbb{Z}$, we actually have a trapdoor one-way *permutation*.

We can think of the RSA game as the one-way function game above, but with F being the RSA function for some specific modulus N and an exponent e. It is believed that for values of N which are a product of two randomly generated primes of roughly the same size that $\mathrm{Adv}^{\mathsf{OWF}}_{\mathsf{RSA}}(A)$ is very small indeed.

11.4. Public Key Cryptography

Recall that in symmetric key cryptography each communicating party needs to have a copy of the same secret key. This leads to a very difficult key management problem, since in a set of n people we would require $n\cdot(n-1)/2$ different symmetric keys to secure all possible communication patterns. In public key cryptography we replace the use of identical keys with two keys, one public and one private, related in a mathematical way.

The public key can be published in a directory along with the user's name. Anyone who then wishes to send a message to the holder of the associated private key will take the public key, encrypt a message under it and send it to that key holder. The idea is that only the holder of the private key will be able to decrypt the message. More clearly, we have the transforms

$$\text{Message} + \text{Alice's public key} = \text{Ciphertext},$$
$$\text{Ciphertext} + \text{Alice's private key} = \text{Message}.$$

Hence anyone with Alice's public key can send Alice a secret message. But only Alice can decrypt the message, since only Alice has the corresponding private key.

Public key systems work because the two keys are linked in a mathematical way, such that knowing the public key does not allow you to compute anything about the private key; think of how knowing N and e tells us nothing about d in the RSA problem. But knowing the private key allows you to unlock information encrypted with the public key; again thinking of the RSA problem, the value d allows us to invert the RSA function.

The concept of being able to encrypt using a key which is not kept secret was so strange it was not until 1976 that anyone thought of it. The idea was first presented in the seminal paper of Diffie and Hellman entitled *New Directions in Cryptography*. Although Diffie and Hellman invented the concept of public key cryptography it was not until a year or so later that the first (and most successful) system, namely RSA, was invented.

The previous paragraph describes the "official" history of public key cryptography. However, in the late 1990s an unofficial history came to light. It turned out that in 1969, over five years before Diffie and Hellman publicly invented public key cryptography, a cryptographer called James Ellis, working for the British government's communication headquarters GCHQ, invented the concept of public key cryptography (or non-secret encryption as he called it) as a means of solving the key distribution problem. Ellis, just like Diffie and Hellman, did not come up with a system.

The problem of finding such a public key encryption system was given to a new GCHQ recruit called Clifford Cocks in 1973. Within a day Cocks had invented what was essentially the RSA algorithm, a full four years before Rivest, Shamir and Adleman invented RSA. In 1974 another

employee at GCHQ, Malcolm Williamson, invented the concept of Diffie–Hellman key exchange, which we shall return to in Chapter 18. Hence, by 1974 the British security services had already discovered the main techniques in public key cryptography.

There are a surprisingly small number of ideas behind public key encryption algorithms, which may explain why, once Diffie and Hellman, and Ellis, had the concept of public key encryption, two inventions of essentially the same cipher (i.e. RSA) came so quickly. There are so few ideas because we require a mathematical operation which is easy to do one way (i.e. encrypt), but which is hard to reverse (i.e. decrypt), without some special secret information, namely the private key. Such a mathematical function is exactly an example of a trapdoor one-way function mentioned above, since it is effectively a one-way function unless one knows the key to the trapdoor. We shall return to methods to construct public key encryption algorithms in a later chapter; for now we are just interested in the abstract concept.

11.5. Security of Encryption

We are now in a position to define the security of both symmetric key and public key encryption algorithms. There are three things we need to define:

- the goal of the adversary,
- the types of attack allowed,
- the computational model.

The first of these corresponds to the winning condition of an analogue of the games considered above (i.e. what does breaking a cryptosystem mean?), the second corresponds to which oracles we allow the adversary to access (i.e. what powers do we give the adversary?), whilst the third is a little more complex, so we will return to it at the end of this chapter.

11.5.1. Basic Notions of Security: To fix notation we assume an encryption scheme is defined for a message space $\mathbb{P}$, a ciphertext space $\mathbb{C}$ and a key space $\mathbb{K}$. For symmetric algorithms we denote the shared key by $k \in \mathbb{K}$. For a public key algorithm we denote the pair of keys by $(\mathfrak{pk}, \mathfrak{sk})$ where $\mathfrak{sk} \in \mathbb{K}$ is the secret key, and $\mathfrak{pk}$ is the associated public key. When generating keys for our games we use the notation $k \leftarrow \text{KeyGen}()$ for symmetric key systems and $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \text{KeyGen}()$ for public key systems.

We denote encryption for symmetric key algorithms by $c \leftarrow e_k(m)$, with decryption given by $m \leftarrow d_k(c)$. For public key algorithms we define encryption and decryption by $c \leftarrow e_{\mathfrak{pk}}(m)$ and $m \leftarrow d_{\mathfrak{sk}}(c)$. We assume that decryption always works for validly encrypted messages, i.e. we have for symmetric key schemes that

$$\forall k \in \mathbb{K},\ \forall m \in \mathbb{P},\ d_k(e_k(m)) = m,$$

with a similar condition holding for public key schemes.

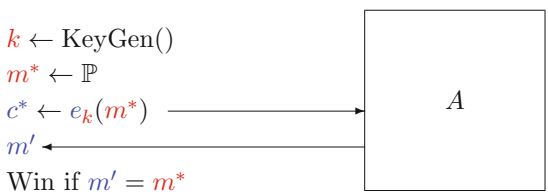


FIGURE 11.6. Security game for symmetric key OW-PASS

As a basic notion of security we take the idea that we do not want an adversary to learn the message underlying a specific ciphertext. This gives us the notion of one-way security, which in

its basic form (for symmetric encryption) is given by the game in Figure 11.6, with the equivalent public key security game being defined by Figure 11.7. We call c^* the *challenge ciphertext* and such an attack is called a *passive attack*. We denote the security game by OW-PASS, as a shorthand for *One Way–PASSive attack*. We define the advantage against the encryption scheme Π by $\mathrm{Adv}_{\Pi}^{\mathsf{OW\text{-}PASS}}(A) = \Pr[A \text{ wins}]$.

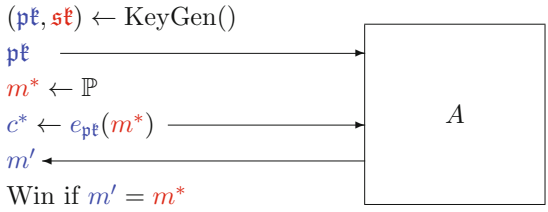


FIGURE 11.7. Security game for public key OW-PASS/OW-CPA

On its own this is a very weak form of security since no oracles are provided to the adversary. In other words the adversary has very limited powers with which she can attack the challenge ciphertext. In particular, in the symmetric key case the adversary is only allowed to see one encrypted message. Thus it is usually the case that the minimum security game also gives the adversary access to an encryption oracle. This attack is called a chosen plaintext attack (CPA), since we imagine giving the adversary access to a (black) box which performs encryption but not decryption on plaintexts of her choosing. See Figure 11.8 for the symmetric key case, defining a notion called OW-CPA (for One Way–Chosen Plaintext Attack). Notice that in the public key passive attack case the adversary already has this ability due to her having the public key; thus the OW-PASS game and OW-CPA game are equivalent in the public key setting. As notation for the advantage we use $\mathrm{Adv}_{\Pi}^{\mathsf{OW\text{-}CPA}}(A) = \Pr[A \text{ wins}]$.

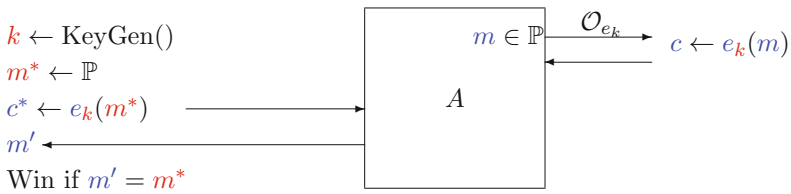


FIGURE 11.8. Security game for symmetric key OW-CPA

A more complex notion of security involves also allowing the adversary the ability to decrypt ciphertexts of her choosing via a decryption oracle. This is called a chosen ciphertext attack (CCA). Clearly, to make the security game non-trivial, we have to stop the adversary requesting the decryption of the challenge ciphertext. A pictorial definition of the OW-CCA attacks in the symmetric and public key settings are given in Figures 11.9 and 11.10. As notation for the advantage we use $\mathrm{Adv}_{\Pi}^{\mathsf{OW\text{-}CCA}}(A) = \Pr[A \text{ wins}]$.

11.5.2. Modern Notions of Security: The above simple notion of "breaking" an encryption algorithm is not very good. In particular it does not allow us to model the situation where an adversary can break a part of a message, but not all of it. Such situations could be important in the real world, and so we need definitions which can define encryption security in the context when the adversary should not be allowed to obtain *any* information about the plaintext. There are essentially three notions of security which we need to understand:

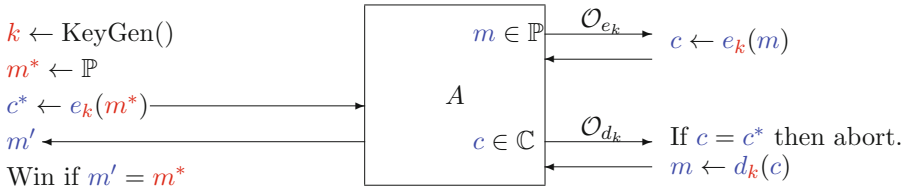


FIGURE 11.9. Security game for symmetric key OW-CCA

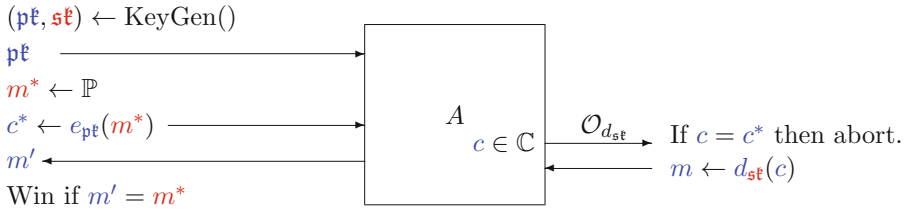


FIGURE 11.10. Security game for public key OW-CCA

- Perfect security,
- Semantic security,
- IND security (short for INDistinguishability of encryptions), a.k.a. polynomial security.

We shall discuss each of these notions in turn. They are far stronger than the simple notions of either recovering the private key or determining the plaintext which we have considered previously.

Perfect Security: Recall that a scheme is said to have perfect security, or information-theoretic security, if an adversary with infinite computing power can learn nothing about the plaintext given the ciphertext. Shannon's Theorem, Theorem 9.4, essentially states that this is achieved if and only if the key is as long as the message, and the same key is never used twice.

The problem is that such systems cannot exist in the public key model, since the encryption key is assumed to be used for many messages (so it is not of one-time use). In addition, for both modern public key and symmetric systems we want to use a short key to encrypt large amounts of data (e.g. a movie). Hence, we will never normally use a system for which the key is as long as the message. Thus, for any notion of security in the real world the notion of perfect security is too strong.

Semantic Security: Semantic security is like perfect security but we only allow an adversary with polynomially bounded computing power. Formally, for all probability distributions on the message space, whatever an adversary can compute (in polynomial time) about the plaintext given the ciphertext, she should also be able to compute without the ciphertext. In other words, having the ciphertext does not help in finding out anything about the message. This is intuitively the equivalent of perfect security for adversaries whose run time is bounded by a polynomial function of the underlying security parameter (i.e. the key size).

The following is a (very) simplified definition which we use purely for illustrative purposes: suppose that the information we wish to compute on the message space is a single bit, i.e. there is some function

$$g : M \longrightarrow \{0, 1\}.$$

We assume that over the whole message space we have

$$\Pr[g(m) = 1] = \Pr[g(m) = 0] = \frac{1}{2},$$

and that the plaintexts and ciphertexts are all of the same length (so in particular the length of the ciphertext reveals nothing about the underlying plaintext).

We model the adversary as an algorithm S which on input of a ciphertext c, encrypted under the symmetric key k, will attempt to produce an evaluation of the function g on the plaintext for which c is the associated ciphertext. The output of S will therefore be a single bit corresponding to the value of g.

The adversary is deemed to be successful if the probability of it producing a correct output is greater than one half. Clearly the adversary could always just guess the bit without seeing the ciphertext, hence we are saying that a successful adversary is one which can do better after seeing the ciphertext. We therefore define the advantage of the adversary S as

$$\mathrm{Adv}_{\Pi}^{\mathsf{SEM}}(S) = 2 \cdot \left| \Pr[S(c) = g(d_k(c))] - \frac{1}{2} \right|.$$

A scheme is then said to be semantically secure if $\mathrm{Adv}_{\Pi}^{\mathsf{SEM}}(S)$ is "small" for all polynomial-time adversaries S[1].

IND Security: The trouble with the definition of semantic security is that it is hard to show that a given encryption scheme has this property. Polynomial security, sometimes called indistinguishability of encryptions, or IND security for short, is a much easier property to confirm for a given system. Luckily, we will show that if a system has IND security then it also has semantic security. Hence, to show that a system is semantically secure all we need do is show that it is IND secure.

A system is said to have indistinguishable encryptions if no adversary can win the following game with probability greater than one half. The adversary will run in two stages:

- **Find**: In the "find" stage the adversary produces two plaintext messages m_0 and m_1, of equal length.
- **Guess**: The adversary is now given the encryption c^* of one of the plaintexts m_b for some secret hidden bit b. The goal of the adversary is to now guess the value of b with probability greater than one half.

Just as for OW security we can define analogous notions of IND-PASS, IND-CPA and IND-CCA for both symmetric key and public key algorithms. We summarize these in Figures 11.11 and 11.12, which give the CCA variants for symmetric and public key encryption respectively. The simpler notions, with fewer oracles, can be derived easily by the reader.

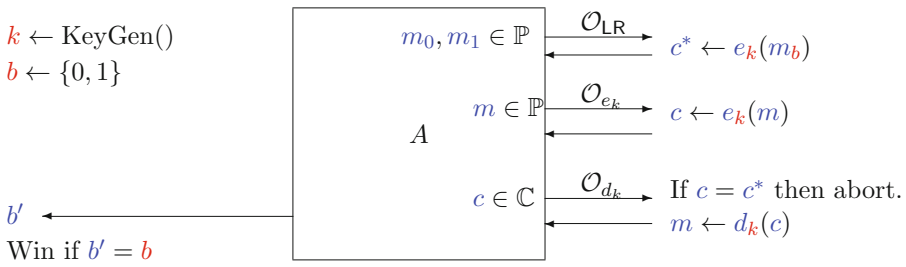


FIGURE 11.11. Security game for symmetric key IND-CCA

[1]Note that this is a very simplified definition of semantic security.

Note that we denote the obtaining of the encryption of m_b via an oracle call, called the LR-oracle (for left–right oracle), which encrypts either the left or right input value depending on b. In the standard IND games the LR-oracle may only be called once by the adversary. We assume in both cases that the adversary may call the encryption and decryption oracles in both the **find** and **guess** phases, i.e. both before and after the call to the $\mathcal{O}_{\mathsf{LR}}$ oracle. If on calling $\mathcal{O}_{\mathsf{LR}}$ the challenger outputs a value c^* which the adversary has already passed to the decryption oracle, then it should also abort. We will implicitly assume this happens in all our diagrams and descriptions.

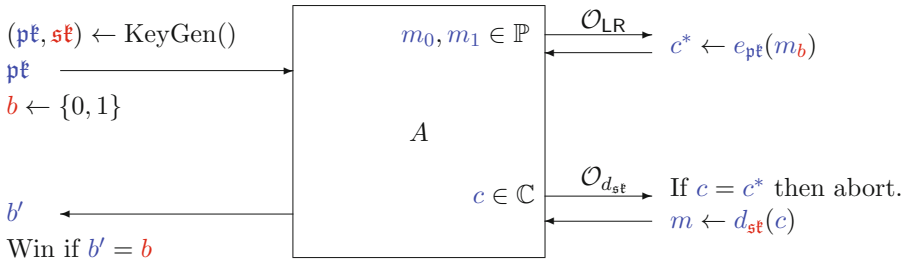


FIGURE 11.12. Security game for public key IND-CCA

Since the adversary A could always simply guess the hidden bit b, we define the advantage of an adversary A just as we did in the PRF games via

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}PASS}}(A) = 2 \cdot \left| \Pr[A \text{ wins}] - \frac{1}{2} \right|.$$

A scheme is then said to be IND-PASS secure if $\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}PASS}}(A)$ is "small" for all polynomial-time adversaries A. Similarly, we can define the advantages $\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CPA}}(A)$ and $\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CCA}}(A)$, and IND-CPA and IND-CCA security.

An important consequence of the above definitions is that *any* encryption function which satisfies IND-CPA security must be probabilistic in nature. If it were not probabilistic then we would have the following attack: In the guess stage the adversary can pass m_1 to her encryption oracle (or encrypt the value itself in the case of public key algorithms) so as to obtain c_1, the encryption of m_1. Then the adversary tests whether c_1 is equal to c^*. Since encryption is deterministic this will determine whether the bit is zero or one.

The following definition is the accepted definition of what it means for an encryption scheme to be secure.

Definition 11.3. *A symmetric key (resp. public key) encryption algorithm is said to be secure if it is* semantically secure *against a chosen ciphertext attack, i.e. for all poly-time adversaries A the value* $\mathrm{Adv}_{\Pi}^{\mathsf{SEM\text{-}CCA}}(A)$ *is "small".*

However, usually it is easier to show security under the following definition.

Definition 11.4. *A symmetric key (resp. public key) encryption algorithm is said to be secure if it is* IND-CCA *secure, i.e. for all poly-time adversaries A the value* $\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CCA}}(A)$ *is "small".*

These two notions are however related. For example, we shall now show the following important result.

Theorem 11.5. *A system which is* IND-PASS *secure must necessarily be semantically secure against passive adversaries.*

PROOF. We proceed by contradiction. Assume that there is an encryption algorithm which is not semantically secure, i.e. that there is an algorithm S with

$$\mathrm{Adv}_{\Pi}^{\mathsf{SEM}}(S) > \epsilon$$

for some "largish" value of ϵ. But we also assume that the encryption algorithm is IND-PASS secure. We shall then derive a contradiction. To do this we construct an adversary A against the IND-PASS security of the scheme, which uses the adversary S against semantic security as an oracle.

The **find** stage of adversary A outputs two messages m_0 and m_1 such that

$$g(m_0) \neq g(m_1).$$

Such messages will be easy to find given our earlier simplified formal definition of semantic security, since the output of g is equiprobable over the whole message space.

The adversary A is then given an encryption c_b of one of these and is asked to determine b. In the **guess** stage the adversary passes the ciphertext c_b to the oracle S. The oracle S returns with its best guess as to the value of $g(m_b)$. The adversary A can now compare this value with $g(m_0)$ and $g(m_1)$ and hence output a guess as to the value of b.

Clearly if S is successful in breaking the semantic security of the scheme, then A will be successful in breaking the polynomial security. So

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}PASS}}(A) = \mathrm{Adv}_{\Pi}^{\mathsf{SEM}}(S) > \epsilon.$$

But such an A is assumed not to exist, since the scheme is IND-PASS secure. This is our contradiction, and hence, IND-PASS security must imply semantic security for passive adversaries. □

Note that with a more complicated definition of semantic security it can be shown that, for adversaries, the notions of semantic and polynomial security are equivalent.

Let Π be a symmetric encryption scheme, then we have the following implications

$$\Pi \text{ is IND-CCA} \implies \Pi \text{ is IND-CPA} \implies \Pi \text{ is IND-PASS},$$

since with each implication moving right we restrict the type of oracles called by the adversary. We also have that

$$\Pi \text{ is IND-XXX} \implies \Pi \text{ is OW-XXX},$$

for XXX equal to PASS, CPA or CCA. For this implication suppose the opposite was true, i.e. that the scheme is IND-XXX secure but not OW-XXX secure. In such a situation we know there is an adversary A which breaks the OW-XXX security of the scheme. It is easy to see that such an adversary can be used to create an adversary B which breaks the IND-XXX security. Let us see how this is done with a picture for the symmetric key case: In Figure 11.13 the adversary B contains A, i.e. it uses A as a subroutine[2]. Clearly B wins the IND-XXX game, and so the implication holds.

The above "proof" is typical of proofs in cryptography. Notice the basic idea: We take an adversary A against one property, and then place it inside another adversary B against another property. We construct the adversary B via "wiring" up adversary A's oracle queries to B's oracles. Note that B has to do this since A is expecting answers to its queries, to which B has to respond. Then we use the answer from A to produce an answer for B. Our assumption is that A exists. However, we have just created B from A. Hence, if we also believe that B does not exist then A cannot exist either. In terms of "code" for the adversary B we have

- $m_0, m_1 \leftarrow \mathbb{P}$.
- Call $\mathcal{O}_{\mathsf{LR}}(m_0, m_1)$ to obtain c^*.
- Call adversary A with target ciphertext c^*.
- If A makes an encryption oracle query, pass the query to B's equivalent oracle and respond to A with the returned value.

[2]We give the diagram for the CCA case. The other cases are left to the reader.

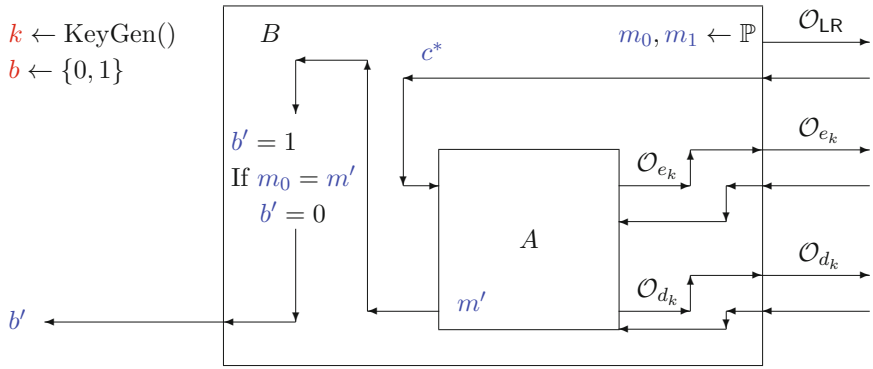


FIGURE 11.13. Constructing an IND-CCA adversary B from a OW-CCA adversary A

- If A makes a decryption oracle query, pass the query to B's equivalent oracle and respond to A with the returned value.
- Eventually A will return a value m'.
- Set $b' = 0$ if $m' = m_0$ and $b' = 1$ otherwise.
- Return b'.

You should convince yourself that the above "code" corresponds to the intuitive description in Figure 11.13, as we will use both code-based and diagrammatically based descriptions as the book progresses. In addition you should also convince yourself that the above "code" creates an algorithm which breaks IND-CCA security given an algorithm which breaks OW-CCA security. In particular we have

$$\text{Adv}^{\text{OW-CCA}}_{\Pi}(A) \leq \text{Adv}^{\text{IND-CCA}}_{\Pi}(B).$$

Thus if Π is IND-CCA secure, i.e. the advantage of B is very small, then the advantage of A must be very small as well. This last statement holds for all adversaries A, as we made no assumption on A bar it was an OW-CCA adverary. Thus Π must also be OW-CCA secure. Hence, as claimed

$$\Pi \text{ is IND-CCA} \implies \Pi \text{ is OW-CCA}.$$

The argument can be extended trivially for CPA and PASS adversaries, as well as the public key case.

11.6. Other Notions of Security

The above are the standard notions of security for encryption schemes. There are many others, some of which we sketch here.

11.6.1. Many Time Security: In the IND game, if we allow the adversary to call the $\mathcal{O}_{\mathsf{LR}}$ oracle many times then we denote the notions by $\text{Adv}^{m\text{-IND-PASS}}_{\Pi}(A)$ etc. Note that we have to modify the decryption oracle in the CCA game so that it aborts if *any* ciphertext returned by the $\mathcal{O}_{\mathsf{LR}}$ oracle is passed to the decryption oracle. If we do not do this then the game can be easily won.

In the symmetric case if we have access to the $\mathcal{O}_{\mathsf{LR}}$ oracle for many calls then we do not need access to the $\mathcal{O}_{e_k}$ oracle, since we can simulate one via the other using the fact that $\mathcal{O}_{e_k}(m) = \mathcal{O}_{\mathsf{LR}}(m, m)$; thus the m-IND-PASS game is equivalent to the m-IND-CPA game[3]. Returning to our implications, we also have that

$$\Pi \text{ is } m\text{-IND-XXX} \implies \Pi \text{ is IND-XXX},$$

[3]In the public key case we never need the $\mathcal{O}_{e_{\mathsf{pk}}}$ oracle in any case.

since an adversary which breaks IND-XXX clearly also trivially breaks m-IND-XXX, by just making a single call to the $\mathcal{O}_{\mathsf{LR}}$ oracle. Thus we have the following theorem.

Theorem 11.6. *Let A be a poly-time adversary against the* IND-XXX *security of the symmetric encryption scheme Π, then there is a poly-time adversary B against the m-*IND-XXX *security of Π, with*

$$\mathrm{Adv}^{\mathsf{IND\text{-}XXX}}_{\Pi}(A) = \mathrm{Adv}^{m\text{-}\mathsf{IND\text{-}XXX}}_{\Pi}(B).$$

Note that this says that if a scheme Π is m-IND-XXX secure, i.e. $\mathrm{Adv}^{m\text{-}\mathsf{IND\text{-}XXX}}_{\Pi}(B)$ is "small" for all poly-time B, then it is also IND-XXX secure, since $\mathrm{Adv}^{\mathsf{IND\text{-}XXX}}_{\Pi}(A)$ will then also be small for all poly-time A by the above theorem. The natural question to ask is whether the other implication holds. Well it does, but not as "tightly" as one would like. One can prove the following result, but it requires a technique called a *hybrid argument* which is a little too advanced for this book.

Theorem 11.7. *Let A be a poly-time adversary against the m-*IND-XXX *security of the symmetric encryption scheme Π, which makes q_{LR} queries to its $\mathcal{O}_{\mathsf{LR}}$ oracle. Then there is a poly-time adversary B against the* IND-XXX *security of Π, with*

$$\mathrm{Adv}^{m\text{-}\mathsf{IND\text{-}XXX}}_{\Pi}(A) \leq q_{\mathsf{LR}} \cdot \mathrm{Adv}^{\mathsf{IND\text{-}XXX}}_{\Pi}(B).$$

Thus security in the IND-XXX game does not directly translate to security in the m-IND-XXX game, as it all depends on how many queries to the $\mathcal{O}_{\mathsf{LR}}$ oracle we allow.

11.6.2. Real-or-Random: In the real-or-random security game the $\mathcal{O}_{\mathsf{LR}}$ oracle is replaced with an $\mathcal{O}_{\mathsf{RoR}}$ oracle. This oracle either encrypts the asked for message (when $b = 1$) or it encrypts a random message of the same length (when $b = 0$). This can either be called once or many times, just as the $\mathcal{O}_{\mathsf{LR}}$ oracle could be called once or many times. This leads us to six new notions of encryption security RoR-XXX and m-RoR-XXX where XXX is one of PASS, CPA or CCA. We present these diagrammatically in Figure 11.14 for the case of symmetric key RoR-CCA. Note that in this case if the $\mathcal{O}_{\mathsf{RoR}}$ oracle can be called many times we cannot use it to replace the $\mathcal{O}_{e_k}$ oracle.

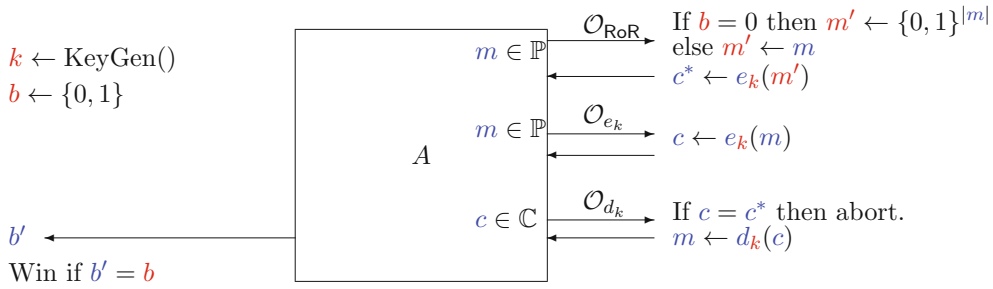


FIGURE 11.14. Security game for symmetric key RoR-CCA

It turns out that RoR security is closely related to IND security. In particular we have the following in the symmetric case (the analogous result can be shown to hold in the public key case).

Theorem 11.8. *A symmetric encryption scheme which is* IND-CCA *secure is also* RoR-CCA *secure. In particular, if A is an adversary against* RoR-CCA *security for a symmetric encryption scheme Π, then we can build an adversary B against* IND-CCA *security of Π with*

$$\mathrm{Adv}^{\mathsf{RoR\text{-}CCA}}_{\Pi}(A) = \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(B).$$

PROOF. Again we construct a proof by wiring the two adversaries together, see Figure 11.15. The inner adversary (subroutine) is A who is able to break the RoR security. From this we construct an adversary B which breaks the IND security of the same encryption scheme Π.

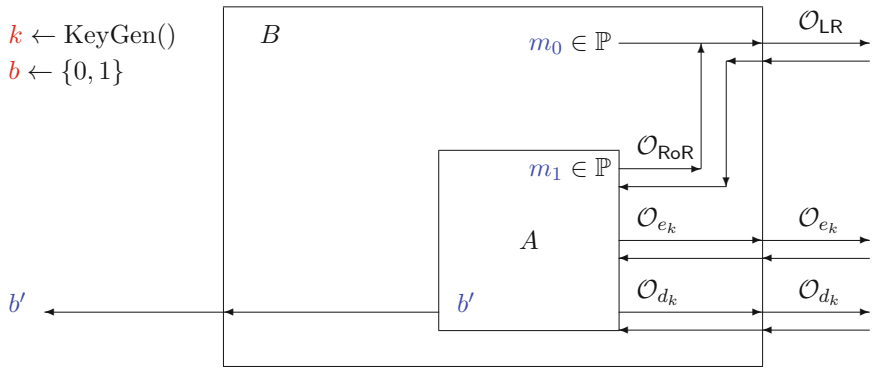


FIGURE 11.15. Constructing an IND-CCA adversary B from an RoR-CCA adversary A (symmetric case)

The advantage statement is immediate since we see that the adversary B is *perfectly simulating* the environment of A, i.e. A cannot tell whether it is playing its game against its normal challenger or is playing its game under the control of the algorithm B. In addition the probability that B wins its game is the same as the probability that A wins its game. For those who find deciphering the diagram difficult, we now give the adversary B in a code-like form:

- Call adversary A.
- If A makes a call to its $\mathcal{O}_{e_k}$ oracle then forward the query to B's $\mathcal{O}_{e_k}$ oracle, and then reply to A with the response from B's oracle.
- If A makes a call to its $\mathcal{O}_{d_k}$ oracle then forward the query to B's $\mathcal{O}_{d_k}$ oracle, and then reply to A with the response from B's oracle.
- When A makes a query to its $\mathcal{O}_{\mathsf{RoR}}$ oracle, take the query m and call it m_1. Generate a random message $m_0 \in \mathbb{P}$ and forward the pair (m_0, m_1) to B's $\mathcal{O}_{\mathsf{LR}}$ oracle. Pass the response back to A to answer her query.
- When A terminates with a bit b', return this as B's answer.

Notice that the $\mathcal{O}_{\mathsf{LR}}$ oracle will return m_1 when $b = 1$, i.e. when the game of A says the real message should be encrypted, and it will return m_0 (a random message chosen by B) when $b = 0$. □

The obvious question which then comes to mind is whether the implication holds the other way around, i.e. is an encryption scheme which is RoR-CCA secure also IND-CCA secure. The answer is perhaps unsurprisingly yes, but what is surprising on first sight is the fact that we "lose" some security as the next result shows.

Theorem 11.9. *A symmetric encryption scheme which is* RoR-CCA *secure is also* IND-CCA *secure. In particular, if A is an adversary against* IND-CCA *security for a symmetric encryption scheme Π, then we can build an adversary B against* RoR-CCA *security of Π with*

$$\text{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) = 2 \cdot \text{Adv}^{\mathsf{RoR\text{-}CCA}}_{\Pi}(B).$$

PROOF. The argument is very similar to the previous case. The reader is invited to draw a picture like that used above. For sake of space we shall only give the code-based description of B.

- Call adversary A.
- If A makes a call to its $\mathcal{O}_{e_k}$ oracle then forward the query to B's $\mathcal{O}_{e_k}$ oracle, and then reply to A with the response from B's oracle.

- If A makes a call to its $\mathcal{O}_{d_k}$ oracle then forward the query to B's $\mathcal{O}_{d_k}$ oracle, and then reply to A with the response from B's oracle.
- When A makes a query to its $\mathcal{O}_{\mathsf{LR}}$ oracle, take the query (m_0, m_1), pick a random bit $t \in \{0,1\}$, and pass m_t to B's $\mathcal{O}_{\mathsf{RoR}}$ oracle. Pass the response back to A to answer its query.
- When A terminates with a bit b', adversary B returns one if $b' = t$ and zero otherwise.

Let us analyse the advantage of B. When the hidden bit $b = 0$, the $\mathcal{O}_{\mathsf{RoR}}$ oracle of B will return a random ciphertext. When we return this to A it has no information about whether m_0 or m_1 was encrypted, because neither were. Thus all that A can do is return a random response b', which will result in B being correct fifty percent of the time.

When the hidden bit $b = 1$ the $\mathcal{O}_{\mathsf{RoR}}$ oracle of B will return the valid encryption of m_t. Thus A will return b', its best guess as to whether $t = 0$ or $t = 1$. So when $b = 1$ we are actually simulating the game for adversary A perfectly.

Picking these two cases apart we obtain

$$\begin{aligned}
\mathrm{Adv}_{\Pi}^{\mathsf{RoR\text{-}CCA}}(B) &= 2 \cdot \left| \Pr[B \text{ wins }] - \frac{1}{2} \right| \\
&= 2 \cdot \left| \Pr[B \text{ wins } \mid b = 1] \cdot \Pr[b = 1] + \Pr[B \text{ wins } \mid b = 0] \cdot \Pr[b = 0] - \frac{1}{2} \right| \\
&= 2 \cdot \left| \Pr[A \text{ wins}] \cdot \Pr[b = 1] + \Pr[b' \neq t] \cdot \Pr[b = 0] - \frac{1}{2} \right| \\
&= 2 \cdot \left| \Pr[A \text{ wins}] \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} \right| \\
&= \left| \Pr[A \text{ wins}] - \frac{1}{2} \right| \\
&= \frac{1}{2} \cdot \mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CCA}}(A).
\end{aligned}$$

□

11.6.3. Lunchtime Attacks: Sometimes, in older literature, the above notion of a CCA attack is called an *adaptive* chosen ciphertext attack, since the chosen ciphertexts passed to the decryption oracle can depend on the challenge ciphertext (but cannot be equal to it of course). This is to distinguish it from an earlier notion called a lunchtime attack, which is sometimes denoted CCA1. In this weaker form of attack the adversary only gets access to the decryption oracle during the "find" stage of the algorithm, i.e. before the $\mathcal{O}_{\mathsf{LR}}$ oracle is called. The lunchtime attack models the situation in which the adversary has access to a black box which performs decryptions, but only at lunchtime whilst the real user of the box is "away at lunch". After this, at some point "in the afternoon", she is given a challenge ciphertext and asked to decrypt it or to learn something about the underlying plaintext, on her own, without using the box.

11.6.4. Nonce-Based Encryption: It should be clear that with the above notions, with the exception of IND-PASS in the symmetric case, an encryption algorithm *must* be randomized. In other words the encryption algorithm, $e_k(m)$ or $e_{\mathtt{pk}}(m)$, must be a randomized algorithm. To see why this is the case consider the following "attack" on a deterministic symmetric scheme.

- Generate m_0 and m_1 of the same length.
- Obtain $c^* \leftarrow e_k(m_b)$ via a call to $\mathcal{O}_{\mathsf{LR}}$.
- Obtain $c \leftarrow e_k(m_0)$ via a call to $\mathcal{O}_{e_k}$.
- If $c = c^*$ then output zero, otherwise output one.

This attack shows us that a deterministic encryption algorithm cannot be IND-CPA secure, since the above adversary runs with probability one of winning.

In practice this is a bit awkward for some encryption algorithms, so we alter the underlying Application Programming Interface (API) slightly and enable the caller of the encryption algorithm to supply the "randomness", as opposed to the randomness coming from inside the encryption algorithm. The only requirement is that the randomness should be of one-time use, and hence should be a *number used once*, or a *nonce* for short. Thus in nonce-based encryption the underlying encryption scheme is assumed to be deterministic, with the "randomness" coming from the nonce only. This notion is mainly used for symmetric encryption algorithms, and a modification of the IND-CCA security game for nonce-based encryption is given in Figure 11.16.

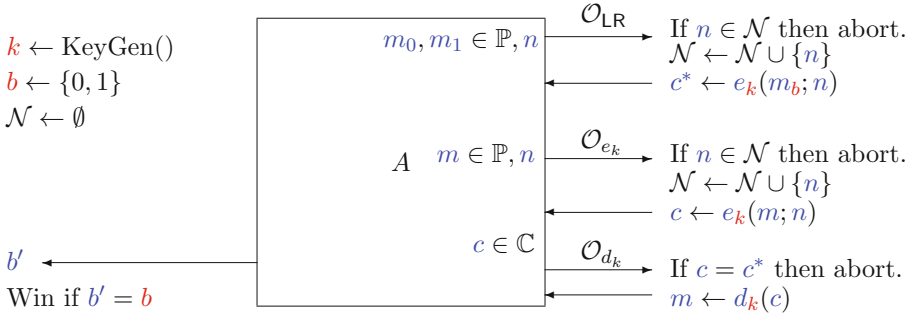


FIGURE 11.16. Security game for nonce-based symmetric key IND-CCA

For the equivalent OW security notion we let the adversary have access to an oracle which will provide a single challenge encryption, using a nonce of the adversary's choice, but for a message uniformly picked from all possible messages of a defined length.

11.6.5. Data Encapsulation Mechanisms: In real-world systems one often uses a fixed symmetric key only once, to encrypt a single message. In this case we do not have a symmetric encryption scheme, but something called a *data encapsulation mechanism*, or DEM for short. Since the attacker against a DEM is such that he can only ever get one ciphertext from the legitimate user we restrict the IND-game for such adversaries to a single call to the $\mathcal{O}_{\mathsf{LR}}$ oracle and no calls to the $\mathcal{O}_{e_k}$ oracle. In the absence of a decryption oracle this is precisely what we have called IND-PASS earlier.

It is sometimes reasonable in such situations to allow the adversary many calls to the $\mathcal{O}_{d_k}$ oracle, since whilst the adversary may only have one ciphertext she could trick a decryptor into trying to decrypt multiple variants of the ciphertext. Such an adversary is called an ot-IND-CCA adversary against the DEM, to signal that the encryption scheme is only being used for one time. The security model is given in Figure 11.17.

Unlike the case of other encryption schemes, as only one ciphertext can ever be obtained, there is no need for the encryption algorithm to be randomized. Thus a DEM is one of the few times that one can use deterministic encryption.

11.6.6. Non-malleability: An important concept related to an encryption scheme is that of *malleability*. An encryption scheme is said to be non-malleable if given a ciphertext c^* corresponding to an unknown plaintext m^*, it is impossible to determine a valid ciphertext c on a 'related' message m. Note that 'related' is defined vaguely here on purpose, but it is assumed that the adversary knows the relation.

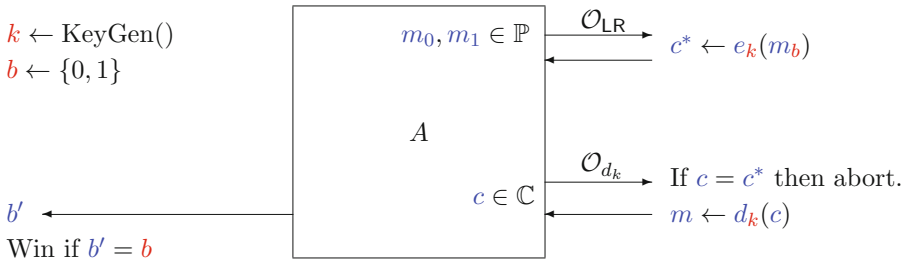


FIGURE 11.17. Security game ot-IND-CCA for a DEM

Thus, in the symmetric key setting a malleability attacker M is one who takes in a ciphertext c^* and who outputs a pair $\{c, f\}$ such that

$$d_k(c) = f\left(d_k(c^*)\right).$$

In the public key setting the definition is the same except that the equation becomes

$$d_{\mathfrak{sk}}(c) = f\left(d_{\mathfrak{sk}}(c^*)\right).$$

In both cases we assume the function f is a non-trivial bijection whose inverse is easy to compute. Non-malleability is important due to the following result, for which we only give an informal proof based on our vague definition of non-malleability. A formal proof can however be given, with an associated formal definition of non-malleability.

Theorem 11.10. *A malleable encryption scheme is not* OW-CCA*, and is hence not* IND-CCA *either.*

PROOF. We give the proof simultaneously in the symmetric and public key settings. Suppose that a scheme is malleable. Our goal is to construct an adversary A against the OW-CCA security of the encryption scheme using the malleability adversary M as a subroutine. Our adversary A takes the challenge ciphertext c^* and passes it to the algorithm M which breaks malleability. We obtain $\{c, f\} \leftarrow M(c^*)$. The adversary then passes the ciphertext c to its decryption oracle. Since $c \neq c^*$ the decryption oracle will return the decryption m of c. We now apply $f^{-1}(m)$ to obtain m^*, thus breaking the OW security of the encryption scheme. □

11.6.7. Plaintext Aware: If a scheme is plaintext aware then we have a very strong notion of security against chosen ciphertext attacks. A scheme is called plaintext aware if it is computationally difficult to construct a valid ciphertext without being given the corresponding plaintext to begin with. Hence, plaintext awareness implies that one cannot mount a CCA attack, since to write down a ciphertext requires you to first know the plaintext, thus making access to the decryption oracle redundant. Thus if a scheme is both IND-CPA and plaintext aware then it will also be IND-CCA. We do not make much use of this notion in this book, so we do not go into a formal game to explain its definition.

11.6.8. Relations Between Security Notions: We have presented a number of different definitions in the above sections for encryption. They are all related as Figure 11.18 presents for some of the cases of symmetric encryption. The values on each arrow relates to the "loss" in security as we move from one security definition to another. As can be seen IND-CCA is a notion which implies all of the others, so it is this notion which we use as the "gold standard" for defining security.

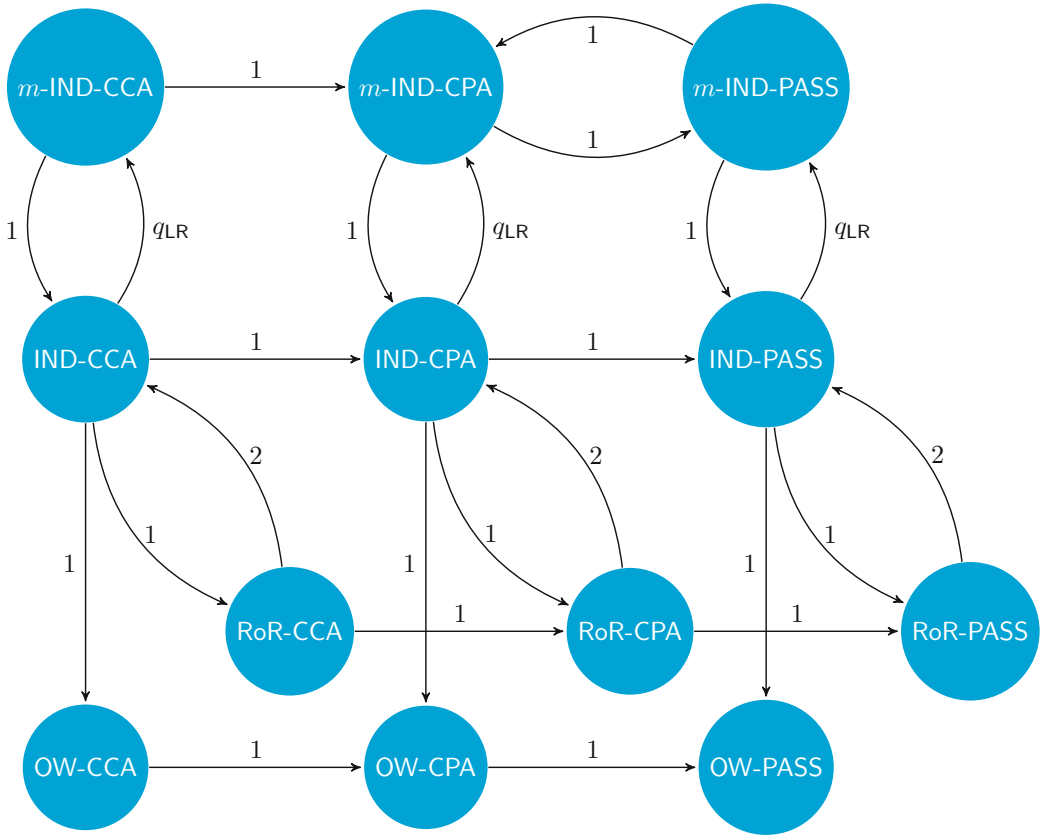


FIGURE 11.18. Relations among notions for symmetric key encryption

11.7. Authentication: Security of Signatures and MACs

Cryptography is more than the production of encryption schemes; a major concern in the subject is that of authentication. How do you know some data is correct (data authentication), and how do you know an entity is who they claim to be (entity authentication). Just as with encryption we can define symmetric and public key algorithms to obtain authentication. In the symmetric setting these are called message authentication codes (or MACs) and in the public key setting these are called digital signatures. We first define these two notions, and then go on to discuss the various security notions.

11.7.1. Message Authentication Codes: Suppose two parties, who share a secret key, wish to ensure that data transmitted between them has not been tampered with. They can then use the shared secret key and a keyed algorithm to produce a tag, or MAC, which is sent with the data. In symbols we compute

$$\text{tag} = \mathsf{Mac}_k(m)$$

where

- Mac is the tag-producing function,
- k is the secret key,
- m is the message.

Note we do not assume that the message is secret: we are trying to protect data integrity and not confidentiality. Producing a tag on its own is not enough; we need a way to verify that it was produced by the correct key.

Hence, we define a MAC scheme as a pair of keyed public algorithms Mac_k and Verify_k such that for a given message m on application of the function Mac_k to m we obtain a value t

$$t \leftarrow \mathsf{Mac}_k(m).$$

The tag is such that when we pass it, and the original message, to the verification algorithm Verify_k, then we will get a bit v signalling a *valid* response.

$$\mathsf{Verify}_k(t, m) = \texttt{valid}.$$

The idea is that it should be hard for the adversary to get such a valid response, unless the tag has been obtained from the MAC algorithm Mac_k using the same key k.

11.7.2. Digital Signature Schemes: Signatures are an important concept of public key cryptography; they also were invented by Diffie and Hellman in the same 1976 paper that invented public key encryption, but the first practical system was due to Rivest, Shamir and Adleman. The basic idea behind public key signatures is as follows:

$$\begin{aligned} \text{Message} + \text{Alice's private key} &= \text{Signature}, \\ \text{Message} + \text{Signature} + \text{Alice's public key} &= \text{YES/NO}. \end{aligned}$$

The above is called a digital signature scheme with appendix. Since the signature is appended to the message before transmission, the message needs to be input into the signature verification procedure. Another variant is the signature scheme with message recovery, where the message is output by the signature verification procedure, as described in

$$\begin{aligned} \text{Message} + \text{Alice's private key} &= \text{Signature}, \\ \text{Signature} + \text{Alice's public key} &= \text{YES/NO} + \text{Message}. \end{aligned}$$

The main idea is that only Alice can sign a message, which could only come from her since only Alice has access to the private key. On the other hand anyone can verify Alice's digital signature, since everyone can have access to her public key. A digital signature scheme consists more formally of two algorithms:

- A signing algorithm S which uses a secret key $\mathfrak{sk}$,
- A verification algorithm V which uses a public key $\mathfrak{pk}$.

In the following discussion, we assume a digital signature with appendix. For a variant with message recovery a simple change to the following will suffice. Alice, sending a message m, calculates

$$s \leftarrow \mathsf{Sig}_{\mathfrak{sk}}(m)$$

and then transmits m, s, where s is the digital signature on the message m. Note that we are not interested in keeping the message secret here, since we are only interested in knowing who it comes from.

The receiver of the signature s applies the verification transform $\mathsf{Verify}_{\mathfrak{pk}}$ to s. The output is a bit v. The bit v indicates valid or invalid, i.e. whether the digital signature is good or not. For correctness we require that

$$\mathsf{Verify}_{\mathfrak{pk}}(\mathsf{Sig}_{\mathfrak{sk}}(m), m) = \texttt{valid},$$

if $\mathfrak{pk}$ is the public key associated with $\mathfrak{sk}$.

11.7.3. Security Definitions for MACs and Digital Signatures: As can be seen from the above definitions there is a great deal of similarity between MACs and digital signatures. We will define their security together, just as we did for symmetric and public key encryption, which will help bring out the similarity in greater detail.

First let us see intuitively what the recipient, Bob, wishes to obtain from the bit v returned by either the MAC or signature verification algorithm when applied to a (message and) tag/signature sent from Alice. There are two main properties

- message integrity – the message has not been altered in transit,
- message origin – the message was sent by Alice,

For digital signatures, since verification is a public operation we can define a third property

- non-repudiation – Alice cannot claim she did not send the message.

To see why non-repudiation is so important, consider what would happen if you could sign a cheque and then say you did not sign it.

The adversary against a MAC or signature algorithm is called a forger. Just like we had OW, IND, RoR etc. notions for encryption algorithms, there are many notions of security for MAC and signature algorithms. The three main types of forgery are:

Total Break: The forger can produce MACs/signatures just as if he were the valid key holder. This is akin to recovering the secret/private key and corresponds to the similar type of break of an encryption algorithm.

Selective Forgery: In this case the adversary is able to forge a MAC/signature on a single message of the challenger's choosing. This is similar to the ability of an adversary of an encryption algorithm being able to decrypt a message but not recover the private key, i.e. like OW security.

Existential Forgery: In this case the adversary is able to forge a MAC/signature on a single message of the adversary's choosing. This message could just be a random bit string, it does not need to mean anything. It can be considered analogous to IND security of encryption schemes.

In practice we usually want our schemes to be secure against an attempt to produce a selective forgery. But we do not know how the MAC/signature scheme is to be used in real life: for example it may be used in a challenge/response protocol where random bit strings are MAC'ed/signed by various parties. Hence, it is prudent to insist that any MAC/signature scheme should be secure against an existential forgery.

Along with types of forgery we also have types of attack. The weakest attack is that of a passive attacker, who is simply given the public key (in the case of a signature algorithm), or a verification oracle (in the case of a MAC algorithm) and is then asked to produce a forgery, be it selective or existential.

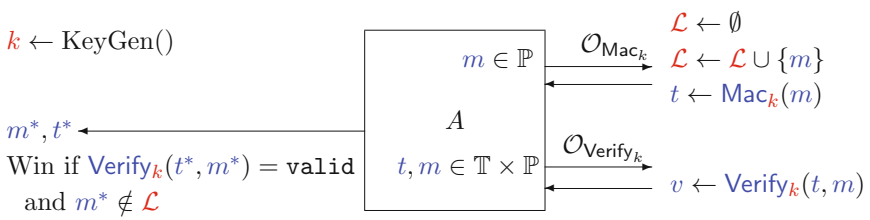


FIGURE 11.19. Security game for MAC security EUF-CMA

The strongest form of attack is that by an adaptive active attacker, who is given access to a MAC/signing oracle which will produce valid MACs/signatures, as well as the data that a passive attacker has access to. The goal of an active attacker is to produce a MAC/signature on a message which she has not yet queried of her MAC/signing oracle. This leads to the following definition:

Definition 11.11. *A MAC/signature scheme is deemed to be secure if it is infeasible for an adaptive adversary to produce an existential forgery. We call this notion* EUF-CMA*, for Existentially UnForgable against a Chosen Message Attack.*

We illustrate these notions in Figures 11.19 and 11.20, where we let $\mathbb{P}$ be the message space, $\mathbb{K}$ the private/secret key space and $\mathbb{T}/\mathbb{S}$ be the tag/signature space.

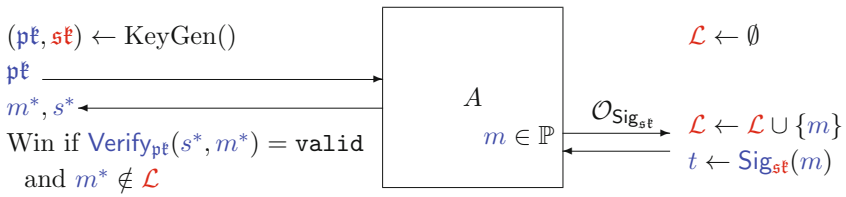


FIGURE 11.20. Security game for signature security EUF-CMA

In terms of advantage statements we define the advantage of an adversary winning the EUF-CMA games (involving either a MAC or signature algorithm Π) as

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\Pi}(A) = \Pr[A \text{ wins the } \mathsf{EUF\text{-}CMA} \text{ game}].$$

So a scheme MAC/signature scheme Π is secure if for all adversaries the associated advantage is "small".

As well as the above notion of forgery there is another notion which is often used in cryptography called *strong* existential unforgeability against a chosen message attack, or sEUF-CMA. This notion applies to both signature and MAC schemes, and in such a security game the winning condition is changed from A winning if she outputs a MAC/signature on a new message, to one where she wins if she outputs a new message/signature pair. We illustrate this for MAC schemes in Figure 11.21

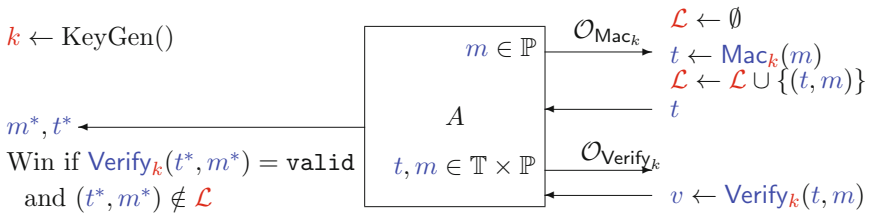


FIGURE 11.21. Security game for MAC security sEUF-CMA

We end this section by noting the following simplification of the MAC and digital signature security games. In many situations a key is only used once by the legitimate party. In the signature setting these are called one-time signatures, however we will also see a use for one-time MACs when we consider the construction of data encapsulation mechanisms later. In such a situation we can limit the adversary to only making *one* query of her MAC/signing oracle, and we call the resulting security notion ot-EUF-CMA.

11.8. Bit Security

Earlier we looked at decision problems, i.e. those problems which output a single bit. An interesting question is whether computing a single bit of the input to a one-way function is as hard as computing the entire input. For example suppose one is using the RSA function

$$F_{N,e} : x \longmapsto y = x^e \pmod{N}.$$

It may be that in a certain system the attacker only cares about computing $b = x \pmod 2$ and not the whole of x. We would like it to be true that computing even this single bit of information about x is as hard as computing all of x. In other words we wish to study the so-called bit security of the RSA function. We can immediately see that bit security is related to semantic security. For example if an attacker could determine the parity of an underlying plaintext given only the ciphertext she could easily break the semantic security of the encryption algorithm.

To formalize this we first define some notation.

Definition 11.12. *Let $f : S \to T$ be a one-way function where S and T are finite sets and let $B : S \to \{0,1\}$ denote a binary function (called a predicate). A hard predicate $B(x)$ for f is one which is easy to compute given $x \in S$ and for which it is hard to compute $B(x)$ given only $f(x) \in T$.*

The way one proves a predicate is a hard predicate, assuming f is a one-way function, is to assume we are given an oracle which computes $B(x)$ given $f(x)$, and then show that this oracle can be used to easily invert f.

A k-bit predicate and hard k-bit predicate are defined in an analogous way but now assuming the codomain of B is the set of bit strings of length k rather than just single bits. We would like to show that various predicates, for given cryptographically useful functions f, are in fact hard predicates.

11.8.1. Hard Predicates for Discrete Logarithms: Let G denote a finite abelian group of prime order q and let g be a generator. Consider the predicate

$$B_2 : x \longmapsto x \pmod 2.$$

We can show the following.

Theorem 11.13. *The predicate B_2 is a hard predicate for the function $x \longmapsto g^x$.*

Proof. Let $\mathcal{O}(h, g)$ denote an oracle which returns the least significant bit of the discrete logarithm of h to the base g, i.e. it computes $B_2(x)$ for $x = \log_g h$. We need to show how to use $\mathcal{O}$ to solve a discrete logarithm problem. Suppose we are given $h = g^x$; we perform the following steps. First we let $t = \frac{1}{2} \pmod q$, then we set $y = 0$, $z = 1$ and compute until $h = 1$ the following steps:

- $b = \mathcal{O}(h, g)$.
- If $b = 1$ then $y = y + z$ and $h = h/g$.
- Set $h = h^t$ and $z = 2 \cdot z$.

We then output y as the discrete logarithm of h with respect to g. □

To see the algorithm in the proof work in practice consider the field $\mathbb{F}_{607}$ and the element $g = 64$ of order $q = 101$. We wish to find the discrete logarithm of $h = 56$ with respect to g. Using the algorithm in the above proof we compute the following table, and hence deduce that x equals 86. One can indeed then check that $g^{86} = h \pmod p$.

h	$\mathcal{O}(h,g)$	z	y
56	0	1	0
451	1	2	2
201	1	4	6
288	0	8	6
100	1	16	22
454	0	32	22
64	1	64	86

11.8.2. Hard Predicates for the RSA Problem: The RSA problem, namely given $c = m^e \pmod N$ has the following three hard predicates:

- $B_1(m) = m \pmod 2$.
- $B_h(m) = 0$ if $m < N/2$ otherwise $B_h(m) = 1$.
- $B_k(m) = m \pmod{2^k}$ where $k = O(\log(\log N))$.

We denote the corresponding oracles by $\mathcal{O}_1(c, N)$, $\mathcal{O}_h(c, N)$ and $\mathcal{O}_k(c, N)$. We do not deal with the last of these but we note that the first two are related since

$$\mathcal{O}_h(c, N) = \mathcal{O}_1(c \cdot 2^e \pmod N, N),$$
$$\mathcal{O}_1(c, N) = \mathcal{O}_h(c \cdot 2^{-e} \pmod N, N).$$

We then have, given an oracle for $\mathcal{O}_h$ or $\mathcal{O}_1$, that we can invert the RSA function using the following algorithm, which is based on the standard binary search algorithm. We let $y = c$, $l = 0$ and $h = N$, then, while $h - l \geq 1$, we perform

- $b = \mathcal{O}_h(y, N)$,
- $y = y \cdot 2^e \pmod N$,
- $m = (h + l)/2$,
- If $b = 1$ then set $l = m$, otherwise set $h = m$.

On exiting the above loop the value of $\lfloor h \rfloor$ should be the preimage of c under the RSA function.

As an example suppose we have $N = 10\,403$ and $e = 7$ as the public information and we wish to invert the RSA function for the ciphertext $c = 3$ using the oracle $\mathcal{O}_h(y, N)$

y	$\mathcal{O}(y, N)$	l	h
3	0	0	10 403
$3 \cdot 2^7$	1	0	5201.5
$3 \cdot 4^7$	1	2600.7	5201.5
$3 \cdot 8^7$	1	3901.1	5201.5
$3 \cdot 16^7$	0	4551.3	5201.5
$3 \cdot 32^7$	0	4551.3	4876.4
$3 \cdot 64^7$	1	4551.3	4713.8
$3 \cdot 128^7$	0	4632.5	4713.8
$3 \cdot 256^7$	1	4632.5	4673.2
$3 \cdot 512^7$	1	4652.9	4673.2
$3 \cdot 1024^7$	1	4663.0	4673.2
$3 \cdot 2048^7$	1	4668.1	4673.2
$3 \cdot 4096^7$	1	4670.7	4673.2
$3 \cdot 8192^7$	0	4671.9	4673.2
-	-	4671.9	4672.5

So the preimage of 3 under the RSA function $x \longmapsto x^7 \pmod{10\,403}$ is 4672.

11.9. Computational Models: The Random Oracle Model

The final item we need to discuss, before we can move on, is the computational model in which we work. When we discussed all the definitions above we assumed something called "the standard model"; this corresponds to the "real world" in some sense. However, there is another model used in cryptography called the Random Oracle Model, or ROM.

To understand this, recall our game for PRF security. Recall that we needed to consider function families since otherwise it is trivial for an adversary to tell whether it is playing against the real function or the random function. If it knows the function precisely then it can call the function itself and see whether the result of the oracle call is real or random. This works for keyed functions like PRF families, but there are many functions in cryptography which are *fixed* yet which we *think* behave like random functions.

In the random oracle model we provide all parties, the challenger and the adversary, with a random function with domain $\{0,1\}^*$ and a finite codomain C. The challenger has control of the random function, whereas the adversary can just call it. In particular the challenger can make up the random function "on the fly", as long as the adversary cannot tell the difference between this function and a random function. Thus an adversary in the ROM has available to it an oracle $\mathcal{O}_H$ as in Figure 11.22

Such a function cannot exist in the real world, as it would take infinite space to represent. However, there are unkeyed cryptographic functions, called hash functions, which are designed to behave like random functions. So in a security proof we may assume that the real unkeyed cryptographic hash function is a truly random function, then when we build the scheme we replace the truly random function by the real unkeyed cryptographic hash function.

What does this mean in practice? We can think of a proof in the ROM as a proof in which the adversary makes no use of knowledge of the real unkeyed cryptographic hash function. In other words from the adversary's point of view the function is just a random function. Thus a proof in the ROM rules out attacks where the adversary makes no use of the specifics of the unkeyed cryptographic hash function. Of course such a proof does not rule out an attack which uses specific properties (for example the code) of the unkeyed cryptographic hash function.

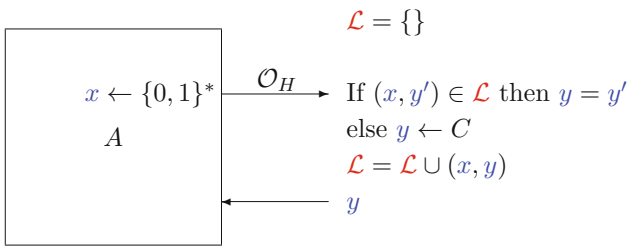


FIGURE 11.22. An adversary with access to a random oracle

The use of the ROM allows us to prove secure various schemes used in the real world. Admittedly, it is a kind of cheat, but a very successful one. We shall return to this and give examples in later chapters, so do not worry at this stage if this looks like an added complication. It actually makes many things much simpler.

Chapter Summary

- We gave definitions of security of pseudo-random functions, pseudo-random permutations, one-way and trapdoor one-way functions.
- We gave the definition of the advantage of an adversary.
- The definition of what it means for a scheme to be secure can be different from one's initial naive view.
- Today the notion of semantic security is the de facto standard definition for encryption schemes.
- Semantic security is hard to prove but it is closely related to the simpler notion of polynomial security, often called indistinguishability of encryptions.
- We also need to worry about the capabilities of the adversary. For encryption this is divided into three categories: passive attacks, chosen plaintext attacks, and chosen ciphertext attacks.
- There are many other notions of encryption security, all of which can be related to each other.
- Encryption security against adaptive adversaries and the notion of non-malleability are closely related.
- Similar considerations apply to the security of signature schemes, where we are now interested in the notion of existential unforgeability under an active attack.
- The Random Oracle Model, or ROM, is a way of creating an idealized random function to help in security analysis.

Further Reading

A good introduction to the definitional work in cryptography based on provable security and its extensions and foundations in the idea of zero-knowledge proofs can be found in the book by Goldreich. A survey of the initial work in this field, up to around 1990, can be found in the article by

Goldwasser. The book by Katz and Lindell takes the approach of definitions and security proofs as its starting point. It is thus a good book to follow up your reading of this one.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Springer, 1999.

S. Goldwasser. *The Search for Provably Secure Cryptosystems*. In Cryptology and Computational Number Theory, Proc. Symposia in Applied Maths, Volume 42, AMS, 1990.

J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. CRC Press, 2007.

CHAPTER 12

Modern Stream Ciphers

Chapter Goals

- To understand the basic principles of modern symmetric ciphers.
- To explain the workings of a modern stream cipher.
- To investigate the properties of linear feedback shift registers (LFSRs).
- To explain how to introduce non-linearity into a stream cipher.

12.1. Stream Ciphers from Pseudo-random Functions

We can interpret a pseudo-random function as a stream cipher. Let $\{F_k\}_K$ be a PRF family with codomain C of bitstrings of length ℓ. The PRF family immediately defines a stream cipher for messages of length ℓ bits. We encrypt a message by setting

$$c = m \oplus F_k(0).$$

We then want to show that this scheme is IND-PASS if the underlying PRF is secure. This result is given in the next theorem.

Theorem 12.1. *If $\{F_k\}_K$ is a PRF family outputting strings of length ℓ bits then the stream cipher Π given by $c = m \oplus F_k(0)$ for $m \in \mathbb{P} = \{0,1\}^\ell$ is* IND-PASS. *In particular*

$$\mathrm{Adv}^{\mathsf{IND\text{-}PASS}}_{\Pi}(A) \leq 2 \cdot \mathrm{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A)$$

Proof. We take the adversary A and consider the game it is playing, as depicted in Figure 12.1. We then change the game slightly, by performing a so-called "game hop". This hop consists of replacing the real PRF function by a completely random function; see Figure 12.2. We call the first game G_0 and the second game G_1. We stress that the adversary (i.e. the algorithm that the adversary runs) in both games is the same; we are just changing the rules of the game.

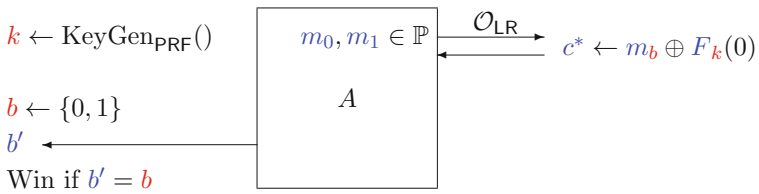


Figure 12.1. Security game G_0 for the scheme $c \leftarrow m \oplus F_k(0)$

Now let b'_0 denote the bit returned by the adversary in game G_0 and b'_1 denote the bit returned by the adversary in game G_1. Similarly let b_0 and b_1 denote the bits chosen by the challenger in games

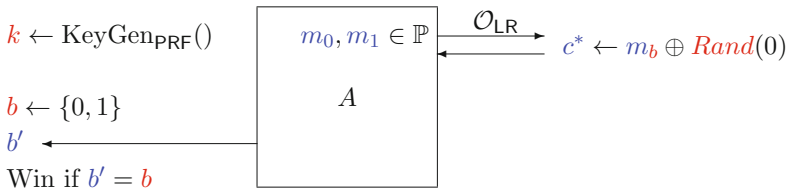


FIGURE 12.2. "Hopped" security game G_1 for the scheme $c \leftarrow m \oplus Rand(0)$

G_0 and G_1 respectively. We have the following relationships between the various probabilities:

$$\left|\Pr[b'_0 = 1 | b_0 = 1] - \Pr[b'_1 = 1 | b_1 = 1]\right| = \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A), \tag{14}$$

and

$$\left|\Pr[b'_0 = 0 | b_0 = 0] - \Pr[b'_1 = 0 | b_1 = 0]\right| = \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A), \tag{15}$$

i.e. for fixed b in both games the difference in the winning probabilities between the two games is the same as the advantage in distinguishing a member of a PRF family from a random function. Also note that

$$\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_1 = 0 | b_1 = 0] = 0 \tag{16}$$

since if we have a random function then an "encryption" of m_0 is a random string, as is an "encryption" of m_1; this is essentially the security of the one-time pad. Thus the probability of the adversary winning in game G_1 is equal to $1/2$. Putting this together we have

$$\begin{aligned}
\text{Adv}^{\mathsf{IND\text{-}PASS}}_{\Pi}(A) &= \left|\Pr[b'_0 = 1 | b_0 = 1] - \Pr[b'_0 = 1 | b_0 = 0]\right| && \text{by definition} \\
&= \Big|\Pr[b'_0 = 1 | b_0 = 1] \\
&\qquad - \big(\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_1 = 1 | b_1 = 1]\big) \\
&\qquad - \Pr[b'_0 = 1 | b_0 = 0]\Big| && \text{adding zero} \\
&\leq \left|\Pr[b'_0 = 1 | b_0 = 1] - \Pr[b'_1 = 1 | b_1 = 1]\right| \\
&\qquad + \left|\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_0 = 1 | b_0 = 0]\right| && \text{triangle inequality} \\
&\leq \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) + \left|\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_0 = 1 | b_0 = 0]\right| && \text{by equation (14)} \\
&= \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) + \Big|\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_0 = 1 | b_0 = 0] \\
&\qquad - \big(\Pr[b'_1 = 0 | b_1 = 0] - \Pr[b'_1 = 0 | b_1 = 0]\big)\Big| && \text{adding zero again} \\
&\leq \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) + \left|\Pr[b'_1 = 1 | b_1 = 1] - \Pr[b'_1 = 0 | b_1 = 0]\right| \\
&\qquad + \left|\Pr[b'_1 = 0 | b_1 = 0] - \Pr[b'_0 = 0 | b_0 = 0]\right| && \text{triangle inequality} \\
&= \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) + \left|\Pr[b'_1 = 0 | b_1 = 0] - \Pr[b'_0 = 0 | b_0 = 0]\right| && \text{by equation (16)} \\
&\leq 2 \cdot \text{Adv}^{\mathsf{PRF}}_{\{F_k\}_K}(A) && \text{by equation (15).}
\end{aligned}$$

□

So when defining a stream cipher we want to look for a candidate which could possibly be a pseudo-random function. In this chapter we will look at various practical constructions of functions which output what looks like random data.

12.2. Linear Feedback Shift Registers

A standard way of producing a binary stream of data is to use a feedback shift register. These are small circuits containing a number of memory cells, each of which holds one bit of information. The set of such cells forms a register. In each cycle a certain predefined set of cells are "tapped" and their value is passed through a function, called the *feedback function*. The register is then shifted down by one bit, with the output bit of the feedback shift register being the bit that is shifted out of the register. The combination of the tapped bits is then fed into the empty cell at the top of the register. Compare this to how we modelled the Lorenz cipher wheels in Chapter 10: the difference is that the output bit is replaced by a new bit which depends on other bits within the register. This is explained in Figure 12.3.

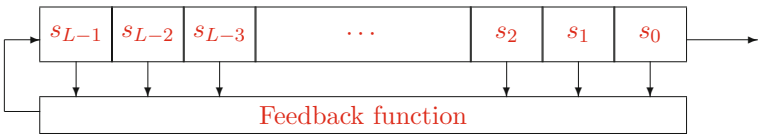


FIGURE 12.3. Feedback shift register

It is desirable, for reasons we shall see later, to use some form of non-linear function as the feedback function. However, this is often hard to do in practice, hence usually one uses a linear feedback shift register, or LFSR for short, where the feedback function is a linear function of the tapped bits. In each cycle a certain predefined set of cells are "tapped" and their value is exclusive-or'ed together. The register is then shifted down by one bit, with the output bit of the LFSR being the bit that is shifted out of the register. Again, the combination of the tapped bits is then fed into the empty cell at the top of the register.

Mathematically this can be defined as follows, where the register is assumed to be of length L. One defines a set of bits $[c_1, \ldots, c_L]$ which are set to one if that cell is tapped and set to zero otherwise. The initial internal state of the register is given by the bit sequence $[s_{L-1}, \ldots, s_1, s_0]$. The output sequence is then defined to be $s_0, s_1, s_2, \ldots, s_{L-1}, s_L, s_{L+1}, \ldots$ where for $j \geq L$ we have

$$s_j = c_1 \cdot s_{j-1} \oplus c_2 \cdot s_{j-2} \oplus \cdots \oplus c_L \cdot s_{j-L}.$$

Note that for an initial state of all zeros the output sequence will be the zero sequence, but for a non-zero initial state the output sequence must eventually be periodic (since we must eventually return to a state we have already been in). The period of a sequence is defined to be the smallest integer N such that

$$s_{N+i} = s_i$$

for all sufficiently large i. In fact there are $2^L - 1$ possible non-zero states and so the most one can hope for is that an LFSR, for all non-zero initial states, produces an output stream whose period is exactly $2^L - 1$.

Each state of the linear feedback shift register can be obtained from the previous state via a matrix multiplication. If we write

$$M = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ c_L & c_{L-1} & c_{L-2} & \dots & c_1 \end{pmatrix}$$

and

$$v = (1, 0, 0, \dots, 0)$$

and we write the internal state as

$$s = (s_1, s_2, \dots, s_L)$$

then the next state can be deduced by computing

$$s \leftarrow M \cdot s$$

and the output bit can be produced by computing the vector product

$$v \cdot s.$$

The properties of the output sequence are closely tied up with the properties of the binary polynomial

$$C(X) = 1 + c_1 \cdot X + c_2 \cdot X^2 + \dots + c_L \cdot X^L \in \mathbb{F}_2[X],$$

called the connection polynomial for the LFSR. The connection polynomial and the matrix are related via

$$C(X) = \det(X \cdot M - I_L).$$

In some textbooks the connection polynomial is written in reverse, i.e. they use

$$G(X) = X^L \cdot C(1/X)$$

as the connection polynomial. One should note that in this case $G(X)$ is the characteristic polynomial of the matrix M.

As examples see Figure 12.4 for an LFSR in which the connection polynomial is given by X^3+X+1 and Figure 12.5 for an LFSR in which the connection polynomial is given by $X^{32}+X^3+1$.

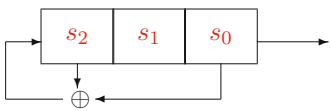


FIGURE 12.4. Linear feedback shift register: $X^3 + X + 1$

FIGURE 12.5. Linear feedback shift register: $X^{32} + X^3 + 1$

Of particular importance is the case in which the connection polynomial is primitive.

Definition 12.2. *A binary polynomial $C(X)$ of degree n is primitive if it is irreducible and a root θ of $C(X)$ generates the multiplicative group of the field $\mathbb{F}_{2^n}$. In other words, since $C(X)$ is irreducible we already have*

$$\mathbb{F}_2[X]/(C(X)) = \mathbb{F}_2(\theta) = \mathbb{F}_{2^n},$$

but we also require $\mathbb{F}_{2^n}^* = \langle\theta\rangle$.

The properties of the output sequence of the LFSR can then be deduced from the following cases.

- $c_L = 0$: i.e. the register is longer than the degree of the connection polynomial.
  In this case the sequence is said to be singular. The output sequence may not be periodic, but it will be eventually periodic.
- $c_L = 1$:
  Such a sequence is called non-singular. The output is always purely periodic, in that it satisfies $s_{N+i} = s_i$ for all i rather than for all sufficiently large values of i. Of the non-singular sequences of particular interest are those satisfying
  - $C(X)$ is irreducible:
    Every non-zero initial state will produce a sequence with period equal to the smallest value of N such that $C(X)$ divides $1 + X^N$. We have that N will divide $2^L - 1$.
  - $C(X)$ is primitive:
    Every non-zero initial state produces an output sequence which is periodic and of exact period $2^L - 1$.

We do not prove these results here, but proofs can be found in any good textbook on the application of finite fields to coding theory, cryptography or communications science. However, we present four examples which show the different behaviours. All examples are on 4-bit registers, i.e. $L = 4$.

Example 1: In this example we use an LFSR with connection polynomial $C(X) = X^3 + X + 1$. We therefore see that $\deg(C) \neq L$, and so the sequence will be singular. The matrix M generating the sequence is given by

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

If we label the states of the LFSR by the number whose binary representation is the state value, i.e. $s_0 = (0, 0, 0, 0)$ and $s_5 = (0, 1, 0, 1)$, then the periods of this LFSR can be represented by the transitions in Figure 12.6. Note that it is not purely periodic.

Example 2: Now let the connection polynomial $C(X) = X^4 + X^3 + X^2 + 1 = (X + 1)(X^3 + X + 1)$, which corresponds to the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

The state transitions are then given by Figure 12.7. Note that it is purely periodic, but with two different cycle lengths due to the different factorization properties of the connection polynomial modulo 2: Two cycles of length $7 = 2^3 - 1$ corresponding to the factor of degree three, and one of length $1 = 2^1 - 1$ corresponding to the factor of degree one. We ignore the trivial cycle of the zero'th state.

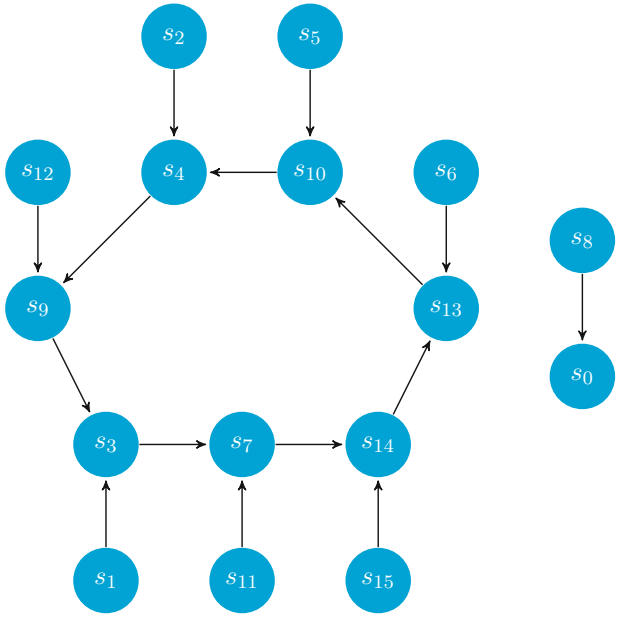


FIGURE 12.6. Transitions of the 4-bit LFSR with connection polynomial $X^3 + X + 1$

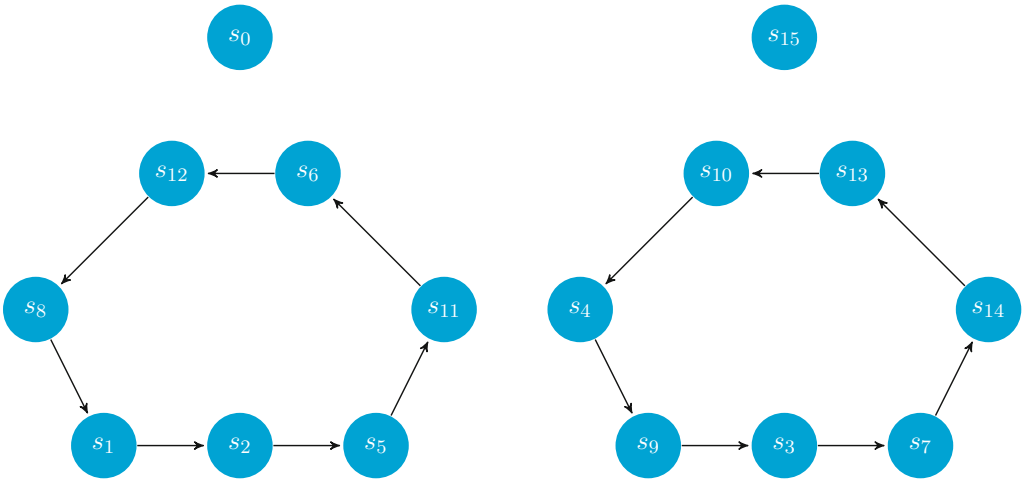


FIGURE 12.7. Transitions of the 4-bit LFSR with connection polynomial $X^4+X^3+X^2+1$

Example 3: Now take the connection polynomial $C(X) = X^4 + X^3 + X^2 + X + 1$, which is irreducible, but not primitive. The matrix is now given by

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The state transitions are then given by Figure 12.8. Note that it is purely periodic and all cycles have the same length, bar the trivial one.

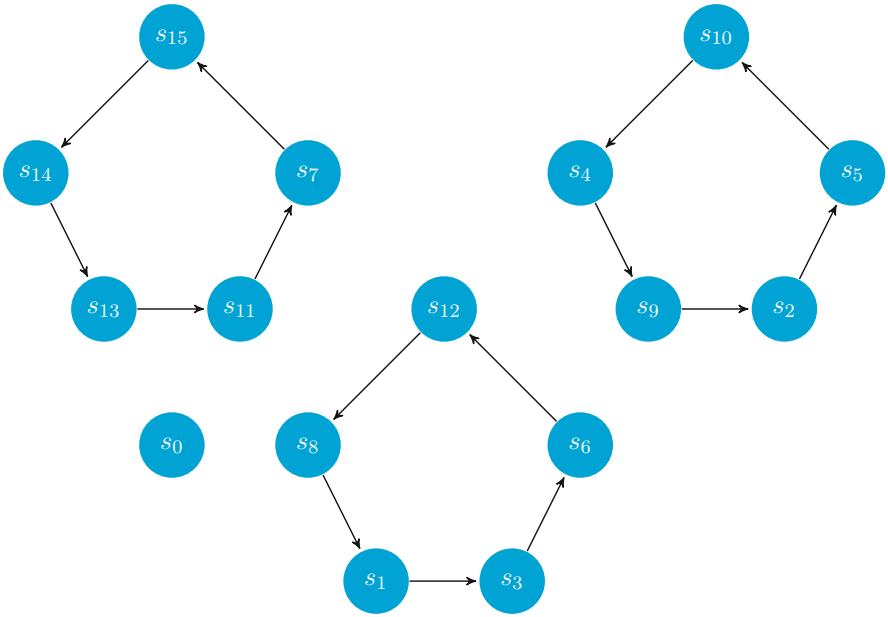


FIGURE 12.8. Transitions of the 4-bit LFSR with connection polynomial $X^4+X^3+X^2+X+1$

Example 4: As our final example we take the connection polynomial $C(X) = X^4 + X + 1$, which is irreducible and primitive. The matrix M is now

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

and the state transitions are given by Figure 12.9.

Whilst there are algorithms to generate primitive polynomials for use in applications we shall not describe them here. The following list gives some examples, all with a small number of taps for efficiency.

$$\begin{array}{ccc} x^{31}+x^3+1, & x^{31}+x^6+1, & x^{31}+x^7+1, \\ x^{39}+x^4+1, & x^{60}+x+1, & x^{63}+x+1, \\ x^{71}+x^6+1, & x^{93}+x^2+1, & x^{137}+x^{21}+1, \\ x^{145}+x^{52}+1, & x^{161}+x^{18}+1, & x^{521}+x^{32}+1. \end{array}$$

Although LFSRs efficiently produce bit streams from a small key, especially when implemented in hardware, they are not usable on their own for cryptographic purposes. This is because they are essentially linear, which is after all why they are efficient.

We shall now show that if we know an LFSR to have L internal registers and we can determine $2 \cdot L$ consecutive bits of the stream then we can determine the whole stream. First notice that

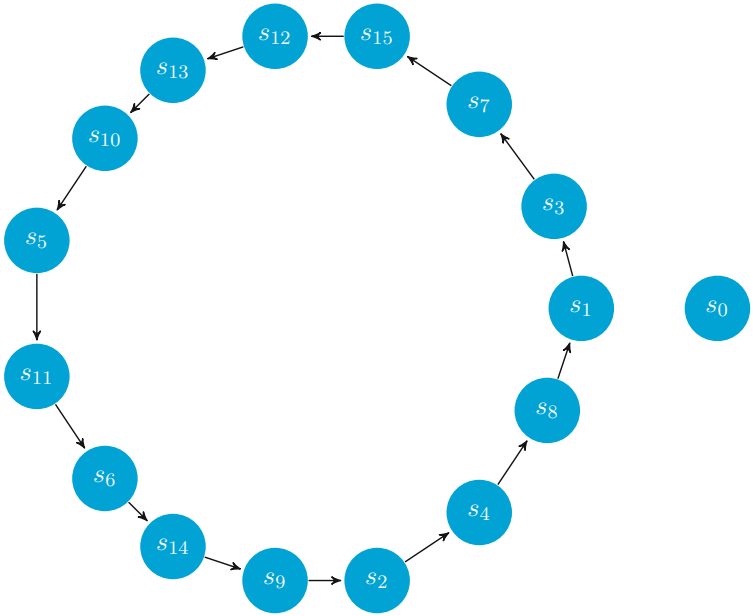


FIGURE 12.9. Transitions of the 4-bit LFSR with connection polynomial $X^4 + X + 1$

we need to determine L unknowns: the L values of the "taps" c_i, since the L values of the initial state $s_0, \ldots, s_{L-1}$ are given to us. This type of data could be available in a known plaintext attack, where we obtain the ciphertext corresponding to a known piece of plaintext; since the encryption operation is simply exclusive-or we can determine as many bits of the keystream as we require. Using the equation

$$s_j = \sum_{i=1}^{L} c_i \cdot s_{j-i} \pmod 2,$$

we obtain $2 \cdot L$ linear equations, which we then solve via standard matrix techniques. We write our matrix equation as

$$\begin{pmatrix} s_{L-1} & s_{L-2} & \cdots & s_1 & s_0 \\ s_L & s_{L-1} & \cdots & s_2 & s_1 \\ \vdots & \vdots & & \vdots & \vdots \\ s_{2L-3} & s_{2L-4} & \cdots & s_{L-1} & s_{L-2} \\ s_{2L-2} & s_{2L-3} & \cdots & s_L & s_{L-1} \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{L-1} \\ c_L \end{pmatrix} = \begin{pmatrix} s_L \\ s_{L+1} \\ \vdots \\ s_{2L-2} \\ s_{2L-1} \end{pmatrix}.$$

As an example, suppose we see the output sequence

$$1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, \ldots$$

and we are told that this sequence was the output of a four-bit LFSR. Using the above matrix equation, and solving it modulo 2, we would find that the connection polynomial was given by

$$X^4 + X + 1.$$

Hence, if we use an LFSR of size L to generate a keystream for a stream cipher and the adversary obtains at least $2 \cdot L$ bits of this keystream then she can determine the exact LFSR used and so generate as much of the keystream as she wishes. Therefore, we would like to be able to adapt

the use of LFSRs in some non-linear way, which hides their linearity in order to produce output sequences with high linear complexity. We can conclude that a stream cipher based solely on a single LFSR is insecure against a known plaintext attack.

12.2.1. Linear Complexity: An important measure of the cryptographic quality of a sequence is given by the linear complexity of the sequence.

Definition 12.3 (Linear complexity)**.** *For an infinite binary sequence*

$$s = s_0, s_1, s_2, s_3, \ldots,$$

we define the linear complexity of s as $L(s)$ where

- $L(s) = 0$ *if s is the zero sequence,*
- $L(s) = \infty$ *if no LFSR generates s,*
- $L(s)$ *is the length of the shortest LFSR to generate s, otherwise.*

Since we cannot compute the linear complexity of an infinite set of bits we often restrict ourselves to a finite set s^n of the first n bits. The linear complexity satisfies the following properties for any sequence s.

- For all $n \geq 1$ we have $0 \leq L(s^n) \leq n$.
- If s is periodic with period N then $L(s) \leq N$.
- $L(s \oplus t) \leq L(s) + L(t)$.

For a random sequence of bits, which is what we want from a stream cipher's keystream generator, we should have that the expected linear complexity of s^n is approximately just larger than $n/2$. But for a keystream generated by an LFSR we know that we will have $L(s^n) = L$ for all $n \geq L$. Hence, an LFSR produces nothing at all like a random bit string. After all it is produced by a linear function!

We have seen that if we know the length of the LFSR then, from the output bits, we can generate the connection polynomial. To determine the length we use the linear complexity profile, which is defined to be the sequence $L(s^1), L(s^2), L(s^3), \ldots$. There is also an efficient algorithm called the Berlekamp–Massey algorithm which given a finite sequence s^n will compute the linear complexity profile

$$L(s^1), L(s^2), L(s^3), \ldots, L(s^n).$$

In addition the Berlekamp–Massey algorithm will also output the associated connection polynomial, if $n \geq L(s^n)/2$, using a technique more efficient than the prior matrix technique.

12.3. Combining LFSRs

To obtain greater security a common practice is to use a number, say n, of LFSRs, each producing a different output sequence $x_1^{(i)}, \ldots, x_n^{(i)}$. The key is then the initial state of all of the LFSRs and the keystream is produced from these n generators using a non-linear combination function $f(x_1, \ldots, x_n)$, as described in Figure 12.10.

We begin by examining the case where the combination function is a Boolean function of the output bits of the constituent LFSRs. For analysis of this function we write it as a sum of distinct products of variables, e.g.

$$f(x_1, x_2, x_3, x_4, x_5) = 1 \oplus x_2 \oplus x_3 \oplus (x_4 \cdot x_5) \oplus (x_1 \cdot x_2 \cdot x_3 \cdot x_5).$$

However, in practice the Boolean function could be implemented in a different way. When expressed as a sum of products of variables we say that the Boolean function is in algebraic normal form.

Suppose that one uses n LFSRs of maximal length (i.e. all with a primitive connection polynomial) and whose periods $L_1, \ldots, L_n$ are all distinct and greater than two. Then, an amazing fact is that the linear complexity of the keystream generated by $f(x_1, \ldots, x_n)$ is equal to

$$f(L_1, \ldots, L_n)$$

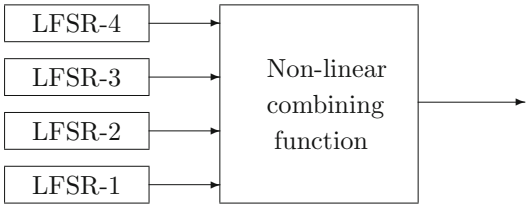


FIGURE 12.10. Combining LFSRs

where we replace $\oplus$ in f with integer addition and multiplication modulo two by integer multiplication, assuming f is expressed in algebraic normal form. The non-linear order of the polynomial f is then defined to be equal to the total degree of f[1].

However, it turns out that creating a non-linear function which results in a high linear complexity is not the whole story. For example, consider the stream cipher produced by the Geffe generator. This generator takes three LFSRs of maximal period and distinct sizes, L_1, L_2 and L_3, and then combines them using the following second-order non-linear function,

$$z = f(x_1, x_2, x_3) = (x_1 \cdot x_2) \oplus (x_2 \cdot x_3) \oplus x_3. \tag{17}$$

This would appear to have very nice properties: its linear complexity is given by

$$L_1 \cdot L_2 + L_2 \cdot L_3 + L_3$$

and its period is given by

$$(2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1).$$

However, it turns out to be cryptographically weak. To understand the weakness of the Geffe generator consider the following table, which presents the outputs x_i of the constituent LFSRs and the resulting output z of the Geffe generator

x_1	x_2	x_3	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If the Geffe generator was using a "good" non-linear combining function then the output bits z would not reveal any information about the corresponding output bits of the constituent LFSRs. However, we can easily see that

$$\Pr(z = x_1) = 3/4 \text{ and } \Pr(z = x_3) = 3/4.$$

This means that the output bits of the Geffe generator are correlated with the bits of two of the constituent LFSRs. Hence, we can attack the generator using a correlation attack, as follows. Suppose we know the lengths L_i of the constituent generators, but not the connection polynomials or their initial states. The attack is desribed in Algorithm 12.1.

[1]The total degree of a polynomial in n variables is the maximum sum of the degrees in each monomial term.

Algorithm 12.1: Correlation attack on the Geffe generator

for all primitive connection polynomials of degree L_1 **do**
  for all initial states of the first LFSR **do**
    Compute $2 \cdot L_1$ bits of output of the first LFSR.
    Compute how many are equal to the output of the Geffe generator.
    A large value signals that this is the correct choice of generator and starting state.
Repeat the above for the third LFSR.
Recover the second LFSR by testing possible values using equation (17).

It turns out that there are a total of

$$S = \phi(2^{L_1} - 1) \cdot \phi(2^{L_2} - 1) \cdot \phi(2^{L_3} - 1)/(L_1 \cdot L_2 \cdot L_3)$$

possible connection polynomials for the three LFSRs in the Geffe generator. The total number of initial states of the Geffe generator is

$$T = (2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1) \approx 2^{L_1+L_2+L_3}.$$

This means that the key size of the Geffe generator is

$$S \cdot T \approx S \cdot (2^{L_1+L_2+L_3}).$$

For a secure stream cipher we would like the size of the key space to be about the same as the number of operations needed to break the stream cipher. However, the above correlation attack on the Geffe generator requires roughly

$$S \cdot (2^{L_1} + 2^{L_2} + 2^{L_3})$$

operations. The reason for the reduced complexity is that we can deal with each constituent LFSR in turn.

To combine high linear complexity and resistance to correlation attacks (and other attacks) designers have had to be a little more ingenious in their choice of non-linear combiners for LFSRs. We now outline a small subset of some of the most influential.

12.3.1. Filter Generator: The basic idea here is to take a single primitive LFSR with internal state $s_1, \ldots, s_L$ and then make the output of the stream cipher a non-linear function of the whole state, i.e. $z = F(s_1, \ldots, s_L)$. If F has non-linear order m then the linear complexity of the resulting sequence is given by

$$\sum_{i=1}^{m} \left(\begin{array}{c} L \\ i \end{array} \right).$$

12.3.2. Alternating-Step Generator: This takes three LFSRs of size L_1, L_2 and L_3 which are pairwise coprime and of roughly the same size. Denote the output sequence of the three LFSRs by x_1, x_2 and x_3. The first LFSR is clocked on every iteration; if its output x_1 is equal to one, then the second LFSR is clocked and the output of the third LFSR is repeated from its last value. If the output of x_1 is equal to zero, then the third LFSR is clocked and the output of the second LFSR is repeated from its last value. The output of the generator is the value of $x_2 \oplus x_3$. This operation is described graphically in Figure 12.11, where (as in Chapter 10) we denote a clocking signal by a black dot to the left of the LFSR which is being clocked. The LFSR will clock one step if the wire has a one on it, and will otherwise remain in its current state. The alternating-step generator has period

$$2^{L_1} \cdot (2^{L_2} - 1) \cdot (2^{L_3} - 1)$$

and linear complexity approximately $(L_2 + L_3) \cdot 2^{L_1}$.

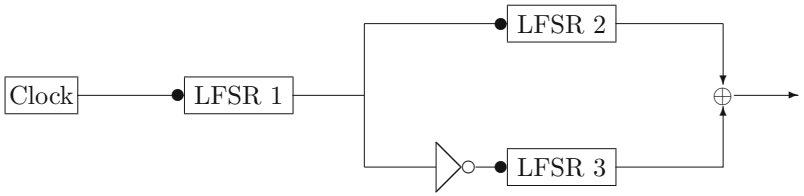


FIGURE 12.11. Graphical representation of the alternating-step generator

12.3.3. Shrinking Generator: Here we take two LFSRs with output sequence x_1 and x_2, and the idea is to throw away some of the x_2 stream under the control of the x_1 stream. Both LFSRs are clocked at the same time, and if x_1 is equal to one then the output of the generator is the value of x_2. If x_1 is equal to zero then the generator just clocks again. Note that, consequently the generator does not produce a bit on each iteration. This operation is described graphically in Figure 12.12. If we assume that the two constituent LFSRs have size L_1 and L_2 with $\gcd(L_1, L_2)$ equal to one, then the period of the shrinking generator is equal to

$$(2^{L_2} - 1) \cdot 2^{L_1 - 1}$$

and its linear complexity is approximately $L_2 \cdot 2^{L_1}$.

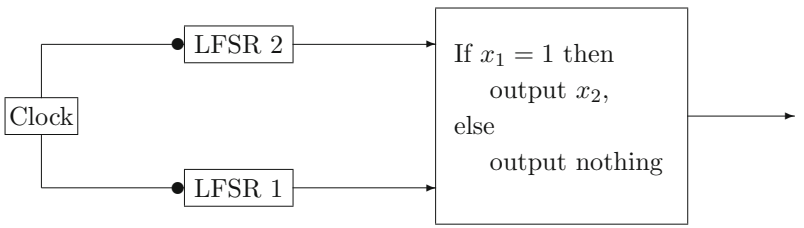


FIGURE 12.12. Graphical representation of the shrinking generator

12.3.4. The A5/1 Generator: Probably the most famous of the LFSR-based stream ciphers is A5/1. This was the stream cipher used to encrypt the on-air traffic in the second generation (a.k.a. GSM) mobile phone networks in Europe and the US. It was developed in 1987, but its design was kept secret until 1999 when it was reverse engineered. There is a weakened version of the algorithm called A5/2 which was designed for use in places to which there were various export restrictions. Various attacks have been published on A5/1 so that it is no longer considered a secure cipher. For example, in 2006 it was shown that one could break into mobile phone conversations which had been protected with A5/1 essentially in real time. In the replacement for GSM, i.e. UMTS (a.k.a. 3G networks) and LTE (a.k.a. 4G networks), the A5/1 cipher has been replaced with the block cipher KASUMI applied in a stream cipher mode of operation.

The stream cipher A5/1 makes use of three LFSRs of lengths 19, 22 and 23. These have characteristic polynomials

$$x^{18} + x^{17} + x^{16} + x^{13} + 1,$$
$$x^{21} + x^{20} + 1,$$
$$x^{22} + x^{21} + x^{20} + x^7 + 1.$$

Alternatively (and equivalently) their connection polynomials are given by

$$x^{18} + x^5 + x^2 + x^1 + 1,$$
$$x^{21} + x^1 + 1,$$
$$x^{22} + x^{15} + x^2 + x^1 + 1.$$

The output of the cipher is the exclusive-or of the three output bits of the three LFSRs.

To clock the registers we associate with each register a "clocking bit". These are in positions 10, 11 and 12 of the LFSRs (assuming bits are ordered with 0 corresponding to the output bit; other books may use a different ordering). We will call these bits c_1, c_2 and c_3. At each clock step the three bits are computed and the "majority bit" is determined via the formulae

$$(c_1 \cdot c_2) \oplus (c_2 \cdot c_3) \oplus (c_1 \cdot c_3).$$

The ith LFSR is then clocked if the majority bit is equal to the bit c_i. Thus clocking occurs subject to the following table.

			Majority	Clock LFSR		
c_1	c_2	c_3	Bit	1	2	3
0	0	0	0	Y	Y	Y
0	0	1	0	Y	Y	N
0	1	0	0	Y	N	Y
0	1	1	1	N	Y	Y
1	0	0	0	N	Y	Y
1	0	1	1	Y	N	Y
1	1	0	1	Y	Y	N
1	1	1	1	Y	Y	Y

We see that in A5/1, each LFSR is clocked with probability 3/4. This operation is described graphically in Figure 12.13, where the "gate" given by an equals sign is the equality-testing gate, and the "gate" labelled by "Maj" is the majority function described above.

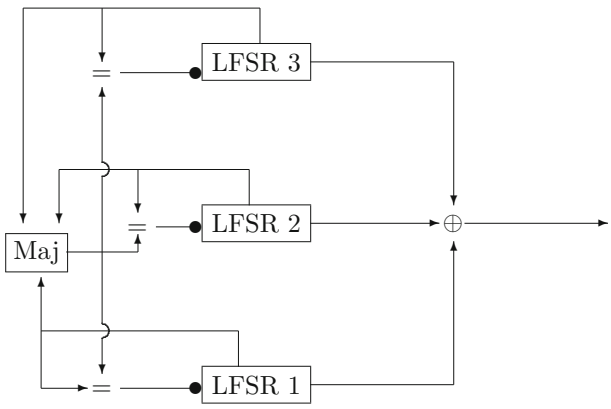


FIGURE 12.13. Graphical representation of the A5/1 generator

12.3.5. Trivium: Trivium is a relatively recent hardware design for a stream cipher, which appears to be more secure than previous designs based on shift registers (although the security of Trivium is not fully guaranteed at this point in time, with some theoretical attacks on it having been presented). The basis of Trivium is a set of three shift registers called a, b and c, of lengths 93, 84 and 111 bits respectively (making 288 bits in total). Once the state has been set up the three shift registers feed into each other via the following equations, over $\mathbb{F}_2$:

$$\begin{aligned} a_i &= c_{i-111} + c_{i-110} + c_{i-109} + c_{i-66} + a_{i-69}, \\ b_i &= a_{i-93} + a_{i-92} + a_{i-91} + a_{i-66} + b_{i-78}, \\ c_i &= b_{i-84} + b_{i-83} + b_{i-82} + b_{i-69} + c_{i-87}. \end{aligned}$$

Notice the regular pattern here: the three top bits of a, b or c are combined with a lower bit (in position 66 or 69) and then with a bit of a second register, to obtain a new bit in the second register. The output bit of Trivium is then obtained from the $\mathbb{F}_2$-equation

$$r_i = c_{i-111} + a_{i-93} + b_{i-84} + c_{i-66} + a_{i-66} + b_{i-84}.$$

To initialize the state an 80-bit key $k_0, \ldots, k_{79}$ and an (up to) 80-bit initial value (IV) $v_0, \ldots, v_{79}$ are fed into the lower bits of the a and b registers, with a getting the key, and b the IV. The rest of the bits of all registers are set to zero, bar the top three bits of the c register. The system is then clocked $4 \cdot 288 = 1152$ times before any keystream is actually used.

Note that this is the first of the stream ciphers we have looked at which explicitly utilizes an IV. We shall see IVs being used in the next chapter on block ciphers, but the basic reason for using them is to move beyond the IND-PASS security of Theorem 12.1. The IV essentially provides a unique input to the keyed PRF that we are trying to produce. So in theoretical terms our cipher becomes $c = m \oplus F_k(\mathrm{IV})$. We do not discuss the theoretical implications here, since much of the discussion on block ciphers in the next chapter will be directly applicable in this situation as well.

12.4. RC4

RC stands for Ron's Cipher after Ron Rivest of MIT. You should not think that the RC4 cipher is a prior version of the block ciphers RC5 and RC6. It is in fact a very, very fast stream cipher. It is easy to remember since it is surprisingly simple. Up until quite recently it was widely deployed in browsers to secure traffic to websites using the TLS protocol. However, recent analysis has shown that the random stream produced by the RC4 algorithm does not behave in a random manner. In particular, each output byte has a particular bias. What is surprising is that the recent analysis is relatively straightforward but the biases had not been discovered in over twenty years of use of the RC4 algorithm. Now that the vulnerability is known, RC4 should no longer be used. However, we present it since it is both historically important and elegantly simple in design.

To describe RC4 we take an array S, indexed from 0 to 255, consisting of the integers $0, \ldots, 255$, permuted in some key-dependent way. The output of the RC4 algorithm is a keystream of bytes K which is exclusive-or'ed with the plaintext byte by byte. Since the algorithm works on bytes and not bits and uses very simple operations, it is particularly fast in software. We start by letting $i = 0$ and $j = 0$. We then repeat the steps in Algorithm 12.2.

Algorithm 12.2: RC4 algorithm

$i \leftarrow (i + 1) \mod 256$.
$j \leftarrow (j + S_i) \mod 256$.
$\mathrm{swap}(S_i, S_j)$.
$t \leftarrow (S_i + S_j) \mod 256$.
$K \leftarrow S_t$.

The security rests on the observation that even if the attacker knows K and i, he can deduce the value of S_t, but this does not allow him to deduce anything about the internal state of the table. This follows from the observation that he cannot deduce the value of t, as he does not know j, S_i or S_j. It is a very tightly designed algorithm as each line of the code needs to be there to make the cipher immune to trivial attacks:

- $i \leftarrow (i+1) \mod 256$**:**
  Makes sure every array element is used once after 256 iterations.
- $j \leftarrow (j+S_i) \mod 256$**:**
  Makes the output depend non-linearly on the array.
- **swap**(S_i, S_j)**:**
  Makes sure the array is evolved and modified as the iteration continues.
- $t \leftarrow (S_i + S_j) \mod 256$**:**
  Makes sure the output sequence reveals little about the internal state of the array.

The initial state of the array S is determined from the key using Algorithm 12.3.

Algorithm 12.3: RC4 key schedule

for $i = 0$ **to** 255 **do** $S_i \leftarrow i$.
Initialize K_i, for $i = 0, \ldots, 255$, with the key, repeating if neccesary.
$j \leftarrow 0$.
for $i = 0$ **to** 255 **do**
  $j \leftarrow (j + S_i + K_i) \mod 256$.
  swap(S_i, S_j).

Chapter Summary

- Many modern stream ciphers can be obtained by combining, in a non-linear way, simple bit generators called LFSRs.
- LFSR-based stream ciphers are very fast ciphers, suitable for implementation in hardware, to encrypt real-time data such as voice or video. But they need to be augmented with a method to produce a form of non-linear output.
- RC4 provides a fast and compact byte oriented stream cipher for use in software, but it is no longer considered secure.

Further Reading

A good introduction to linear recurrence sequences over finite fields is in the book by Lidl and Niederreiter. This book covers all the theory one requires, including examples and a description of the Berlekamp–Massey algorithm. The attacks on the A5/1 algorithm are described in the paper by Barkan et al. The paper by AlFardan et al. covers recent analysis of the RC4 stream cipher.

N.J. AlFardan, D.J. Bernstein, K.G. Paterson, B. Poettering and J.C.N. Schuldt. *On the security of RC4 in TLS*. USENIX Security Symposium, 305–320, USENIX Association, 2013.

E. Barkan, E. Biham and N. Keller. *Instant ciphertext-only cryptanalysis of GSM encrypted communication*. Journal of Cryptology, **21**, 391–429, 2008.

R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1986.

CHAPTER 13

Block Ciphers and Modes of Operation

Chapter Goals

- To introduce the notion of block ciphers.
- To understand the workings of the DES algorithm.
- To understand the workings of the AES algorithm.
- To learn about the various standard modes of operation of block ciphers.

13.1. Introduction to Block Ciphers

The basic description of a block cipher is shown in Figure 13.1. Block ciphers operate on blocks

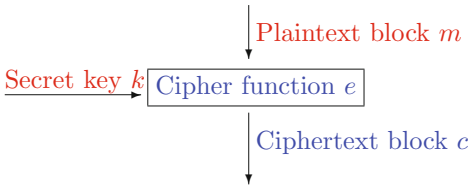


FIGURE 13.1. Operation of a block cipher

of plaintext one at a time to produce blocks of ciphertext. The block of plaintext and the block of ciphertext are assumed to be of the same size, e.g. a block of n bits. Every string of n bits in the domain should map to a string of n bits in the codomain, and every string of n bits in the codomain should result from the application of the function to a string in the domain. This means that for a fixed key a block cipher is bijective and hence is a permutation. We write

$$c \leftarrow e_k(m),$$
$$m \leftarrow d_k(c)$$

where

- $m \in \{0,1\}^n$ is the plaintext block,
- $k \in K$ is the secret key, chosen from key space K,
- e is the encryption function,
- d is the decryption function,
- $c \in \{0,1\}^b$ is the ciphertext block.

The block sizes taken are usually reasonably large, 64 bits in DES and 128 bits or more in modern block ciphers.

In terms of our prior definitions from Chapter 11, a block cipher should "act like" a family of pseudo-random permutations (PRPs), indexed by the key space K, $\{F_k\}_K$. We put the "act like"

in quotes, as for all existing (efficient) block ciphers used in practice we cannot mathematically prove that they are PRPs. Thus we can only "hope" that no adversary can break the PRP security game (Figure 11.4 of Chapter 11) for the specific block cipher[1]. In particular we hope that the advantage of an adversary against the PRP property is something like

$$\mathrm{Adv}^{\mathsf{PRP}}_{\{F_k\}_K}(A) \approx 1/|K|$$

for all adversaries A. So we require the key space to be rather large to ensure this advantage is small. But note that just because a block cipher has a large key space does not mean it will be a secure PRP.

Despite its name a block cipher *is not* an encryption scheme; it is a building block to create an encryption scheme. The term for how one creates an encryption scheme out of a block cipher is a *mode of operation*. The key advantage of this division, between a mode of operation and a block cipher design, is that we can design our modes and our block ciphers independently. As remarked, the design goal for a block cipher is that it is a secure pseudo-random permutation, whereas the design goal of a mode of operation is one of our security goals, such as IND-CCA. A designer of a mode of operation tries to prove mathematically that the mode satisfies the required security definition on the assumption that the block cipher is a secure PRP.

There are many block ciphers in use today, some which you may find used in your web browser; these include AES, CAMELLIA, DES or 3DES. The most famous of these is DES, or the Data Encryption Standard. This was first published in the mid-1970s as a US Federal standard and soon became the de facto international standard for banking applications.

The DES algorithm stood up remarkably well to the test of time, but in the early 1990s it became clear that a new standard was required. This was because both the block length (64 bits) and the key length (56 bits) of basic DES were too small for new applications. It is now possible to recover a 56-bit DES key using either a network of computers or specialized hardware for relatively little cost. Therefore DES has been phased out of most applications; although it still exists as a component in the variant called triple DES (3DES). In response to the problem of DES being deemed insecure, the US National Institute of Standards and Technology (NIST) initiated a competition to find a new block cipher, to be called the Advanced Encryption Standard or AES.

Unlike the process used to design DES, which was kept essentially secret, the design of the AES was performed in public. A number of groups from around the world submitted designs for the AES. Eventually five algorithms, known as the AES finalists, were chosen to be studied in depth. These were

- MARS from a group at IBM,
- RC6 from a group at RSA Security,
- Twofish from a group based at Counterpane, UC Berkeley and elsewhere,
- Serpent from a group of three academics based in Israel, Norway and the UK,
- Rijndael from a couple of Belgian cryptographers.

Finally in the fall of 2000, NIST announced that the overall AES winner had been chosen to be Rijndael, and so from hence forth Rijndael was known as AES.

DES and all the AES finalists are examples of *iterated* block ciphers. Block ciphers obtain their security by repeated use of a simple *round function*. The round function takes an n-bit block and returns an n-bit block, where n is the block size of the overall cipher. The number of rounds r can either be variable or fixed. As a general rule increasing the number of rounds will increase the level of security of the block cipher.

[1] An interesting side effect of our constructions is that we make no assumption on whether it is hard to break the block cipher given an oracle for the PRP in *both* the forwards or backwards directions. After reading this chapter you might want to consider why this is.

Each use of the round function employs a round key k_i for $1 \leq i \leq r$ derived from the main secret key k, using an algorithm called a *key schedule*. To allow decryption, for every round key the function implementing the round must be invertible, and for decryption the round keys are used in the opposite order to that in which they were used for encryption. That the whole round is invertible does not imply that the functions used to implement the round need to be invertible. This may seem strange at first reading but will become clearer when we discuss the DES cipher later. In DES the functions needed to implement the round function are not invertible, but the whole round is invertible. For AES not only is the whole round function invertible but every function used to create the round function is also invertible.

There are a number of general-purpose techniques which can be used to break a block cipher, for example: exhaustive search, using pre-computed tables of intermediate values or divide and conquer. Some (badly designed) block ciphers can be susceptible to chosen plaintext attacks, where encrypting a specially chosen plaintext can reveal properties of the underlying secret key. In cryptanalysis one needs a combination of mathematical and puzzle-solving skills, plus luck. There are a few more advanced techniques which can be employed:

- **Differential Cryptanalysis:** In differential cryptanalysis one looks at ciphertext pairs, where the corresponding plaintexts have a particular difference. The exclusive-or of such pairs is called a differential and certain differentials have certain probabilities associated with them, depending on what the key is. By analysing the probabilities of the differentials computed in a chosen plaintext attack one can hope to reveal the underlying structure of the key.
- **Linear Cryptanalysis:** Even though a good block cipher should contain non-linear components the idea behind linear cryptanalysis is to approximate the behaviour of the non-linear components with linear functions. Again the goal is to use a probabilistic analysis to determine information about the key.

Surprisingly these two methods are quite successful against some ciphers. Both DES and AES are designed to resist differential cryptanalysis, whereas AES is designed to also resist linear cryptanalysis.

Since DES and AES are likely to be the most important block ciphers in use for the next few years we shall study them in some detail. We also do this as they both show general design principles in their use of substitutions and permutations. Recall that the historical ciphers in Chapter 7 made use of such operations, so we see that not much has changed. Now, however, the substitutions and permutations used are far more intricate. On their own they do not produce security, but when used over a number of rounds one can obtain enough security for our applications.

We end this section by discussing the question, which is best, a block cipher or a stream cipher? The main difference between a block cipher and a stream cipher is that block ciphers are stateless, whilst stream ciphers maintain an internal state which is needed to determine which part of the keystream should be generated next. Here are just a few general points.

- Block ciphers are more general, and we shall see that one can easily turn a block cipher into a stream cipher.
- Stream cipher designs generally have a more mathematical structure. This either makes them easier to break or easier to study to convince oneself that they are secure.
- Stream ciphers are generally not suitable for software, since they usually encrypt one bit at a time. However, stream ciphers are highly efficient in hardware.
- Block ciphers are suitable for both hardware and software, but are generally not as fast in hardware as stream ciphers.

- Hardware is always faster than software, but this performance improvement comes at the cost of less flexibility.
- One can use block ciphers to build more complex functions via modes of operation, and rigorously analyse them if we assume the block cipher is a secure PRP.

13.2. Feistel Ciphers and DES

The DES cipher is a variant of the basic Feistel cipher described in Figure 13.2. Feistel ciphers are named after H. Feistel, who worked at IBM and performed some of the earliest non-military research on encryption algorithms. The interesting property of a Feistel cipher is that the round function is invertible regardless of the choice of the function in the box marked F. To see this notice that each encryption round is given by

$$L_i \leftarrow R_{i-1},$$
$$R_i \leftarrow L_{i-1} \oplus F(K_i, R_{i-1}).$$

Hence, the decryption can be performed via

$$R_{i-1} \leftarrow L_i,$$
$$L_{i-1} \leftarrow R_i \oplus F(K_i, L_i).$$

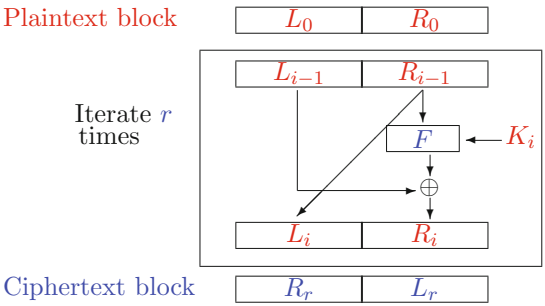


FIGURE 13.2. Basic operation of a Feistel cipher

This means that in a Feistel cipher we have simplified the design somewhat, since

- we can choose any function for the function F, and we will still obtain an encryption function which can be inverted using the secret key,
- the same code/circuitry can be used for the encryption and decryption functions. We only need to use the round keys in the reverse order for decryption.

Of course to obtain a secure cipher we still need to take care with

- how the round keys are generated,
- how many rounds to take,
- how the round function F is defined.

Work on DES was started in the early 1970s by a team in IBM which included Horst Feistel. It was originally based on an earlier cipher of IBM's called Lucifer, but some of the design was known to have been amended by the National Security Agency (NSA). For many years this led conspiracy theorists to believe that the NSA had placed a trapdoor in the design of the function F. However, it is now widely accepted that the modifications made by the NSA were done to make the cipher

more secure. In particular, the changes made by the NSA made the cipher resistant to differential cryptanalysis, a technique that was not discovered in the open research community until the 1980s.

DES is also known as the Data Encryption Algorithm (DEA) in documents produced by the American National Standards Institute (ANSI). The International Organization for Standardization (ISO) refers to DES by the name DEA-1. It was a worldwide standard for around thirty years and stands as the first publicly available cryptographic algorithm to have an "official status". It therefore marks an important step on the road from cryptography being a purely military area to being a tool for the masses. The use of DES is now no longer recommended on its own, and it has been withdrawn from all standards.

The basic properties of the DES cipher are that it is a variant of the Feistel cipher design in which

- the number of rounds r is 16,
- the block length n is 64 bits,
- the key length is 56 bits,
- the round keys $K_1, \ldots, K_{16}$ are each 48 bits.

Note that a key length of 56 bits is insufficient for many modern applications, hence often one uses DES by using three keys and three iterations of the main cipher. Such a version is called triple DES or 3DES; see Figure 13.3.

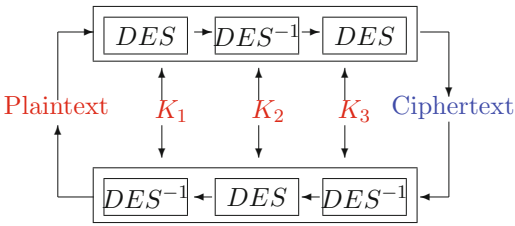


FIGURE 13.3. Triple DES

In 3DES the key length is equal to 168. There is another way of using DES three times, but using two keys instead of three giving rise to a key length of 112. In this two-key version of 3DES, one uses the 3DES basic structure but with the first and third key being equal. However, two-key 3DES is not as secure as one might inititally think. Intuitively one might suspect that it has a key size of 112 bits, however one can break it with 2^{64} effort.

Theorem 13.1. *Two-key 3DES can be broken in about 2^{64} time and about 2^{64} space, with a chosen plaintext attack.*

PROOF. The two-key variant of 3DES is given by the equation

$$c \leftarrow DES_{k_1}(DES_{k_2}{}^{-1}(DES_{k_1}(m))).$$

The technique to break this variant is a standard time/memory trade-off algorithm, which is very similar to the Baby-Step/Giant-Step algorithm of Chapter 3.

The attacker executes the following steps:

(1) For all $t_i \in K$ we compute $a_i \leftarrow DES_{t_i}{}^{-1}(0)$, where 0 is the all-zero message block,
(2) We store the 2^{64} tuples (a_i, t_i) in a table.
(3) For each tuple, we submit each a_i as a *plaintext* to our chosen plaintext attack oracle. We obtain
$$c_i \leftarrow DES_{k_1}(DES_{k_2}{}^{-1}(DES_{k_1}(a_i))).$$
(4) For each value c_i we compute $b_i \leftarrow DES_{t_i}{}^{-1}(c_i)$.

(5) Using the table, we look for pairs (a_j, b_i) for which $a_j = b_i$; this can be done fast for each b_i by sorting or hashing the initial table.
(6) Output (t_i, t_j) as a possible key, which can be tested with a few further chosen plaintext attack oracle queries.

To see why this attack works, consider the following series of identities,

$$\begin{aligned}
DES_{t_i}(DES_{t_j}{}^{-1}(DES_{t_i}(a_i))) &= DES_{t_i}(DES_{t_j}{}^{-1}(DES_{t_i}(DES_{t_i}{}^{-1}(0)))) && \text{by step one}\\
&= DES_{t_i}(DES_{t_j}{}^{-1}(0)) && \text{by definition of DES}\\
&= DES_{t_i}(a_j) && \text{by step one}\\
&= DES_{t_i}(b_i) && \text{by step five}\\
&= DES_{t_i}(DES_{t_i}{}^{-1}(c_i)) && \text{by step four}\\
&= c_i && \text{by definition of DES.}
\end{aligned}$$

This is exactly what the chosen plaintext oracle outputs. Thus it is highly likely that $(k_1, k_2) = (t_i, t_j)$, which can be confirmed by encrypting a few more plaintexts as in step six. □

A similar time/memory trade-off can be applied to the full 3DES algorithm. However, the result is that the effective complexity is 2^{112}. Thus 3DES is still considered secure, just not as secure as one would expect from its key size of 168 bits, but we have the expectation that

$$\mathrm{Adv}^{\mathsf{PRP}}_{\{3DES_k\}_K}(A) \approx 1/2^{112}.$$

13.2.1. Overview of DES Operation: Basically DES is a Feistel cipher with 16 rounds, except that before and after the main Feistel iteration a permutation is performed, as depicted in Figure 13.4. This permutation appears to produce no change in the security, and people have often wondered why it is there. One answer given by one of the original team members was that this permutation was there to make the original implementation easier to fit on the circuit board.

In summary the DES cipher operates on 64 bits of plaintext in the following manner:

- Perform an initial permutation.
- Split the blocks into left and right half.
- Perform 16 rounds of identical operations.
- Join the half blocks back together.
- Perform a final inverse permutation.

The final permutation is the inverse of the initial permutation; this allows the same hardware and/or software to be used for encryption and decryption. The key schedule provides 16 round keys of 48 bits in length by selecting 48 bits from the 56-bit main key. We shall now describe the operation of the function F in more detail. In each DES round this consists of the following six stages:

- **Expansion Permutation:** The right half of 32 bits is expanded and permuted to 48 bits. This helps the diffusion of any relationship of input bits to output bits. The expansion permutation (which is different from the initial permutation) has been chosen so that one bit of input affects two substitutions in the output, via the S-Boxes below. This helps spread dependencies and creates an avalanche effect (a small difference between two plaintexts will produce a very large difference in the corresponding ciphertexts).
- **Round Key Addition:** The 48-bit output from the expansion permutation is exclusive-or'd with the round key, which is also 48 bits in length. Note that this is the only place where the round key is used in the algorithm.
- **Splitting:** The resulting 48-bit value is split into eight lots of six-bit values.

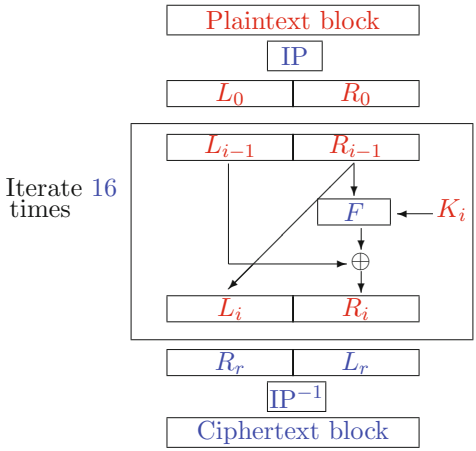


FIGURE 13.4. DES as a Feistel cipher

- **S-Boxes:** Each six-bit value is passed into one of eight different S-Boxes (Substitution Box) to produce a four-bit result. The S-Boxes represent the non-linear component in the DES algorithm and their design is a major contributor to the algorithm's security. Each S-Box is a look-up table of four rows and sixteen columns. The six input bits specify which row and column to use. Bits 1 and 6 generate the row number, whilst bits $2, 3, 4$ and 5 specify the column number. The output of each S-Box is the value held in that element in the table.
- **P-Box:** We now have eight lots of four-bit outputs which are then combined into a 32-bit value and permuted to form the output of the function F.

The overall structure of the DES F function is explained in Figure 13.5.

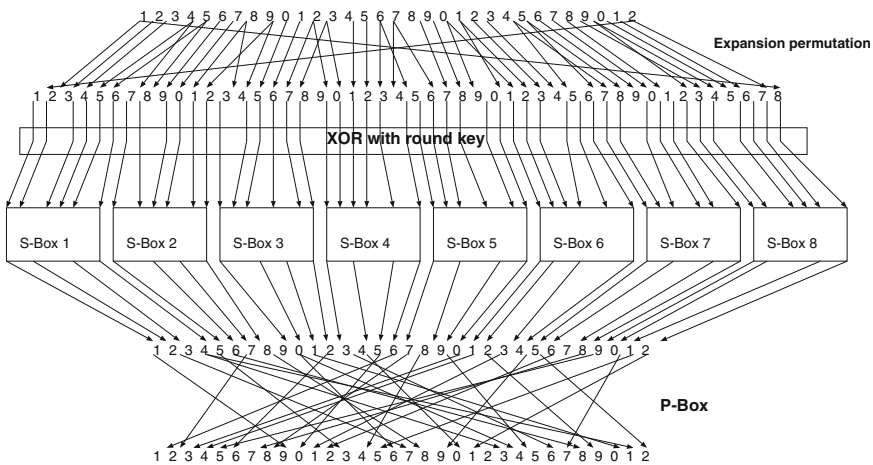


FIGURE 13.5. Structure of the DES function F

We now give details of each of the steps which we have not yet fully defined.

Initial Permutation IP: The DES initial permutation is defined in the following table. Here the 58 in the first position means that the first bit of the output from the IP is the 58th bit of the input, and so on.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

The inverse permutation is given in a similar manner by the following table.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Expansion Permutation E: The expansion permutation is given in the following table. Each row corresponds to the bits which are input into the corresponding S-Box at the next stage. Notice how the bits which select the row of one S-Box (the first and last bit on each row) are also used to select the column of another S-Box.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

S-Box: The details of the eight DES S-Boxes are given in Figure 13.6. Recall that each box consists of a table with four rows and sixteen columns.

The P-Box Permutation P: The P-Box permutation takes the eight lots of four-bit nibbles, output by the S-Boxes, and produces a 32-bit permutation of these values as given by the following table.

S-Box 1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-Box 2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-Box 3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S-Box 4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-Box 5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-Box 6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-Box 7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-Box 8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

FIGURE 13.6. DES S-Boxes

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

DES Key Schedule: The DES key schedule takes the 56-bit key, which is actually input as a bitstring of 64 bits comprising of the key and eight parity bits, for error detection. These parity bits are in bit positions $8, 16, \ldots, 64$ and ensure that each byte of the key contains an odd number of bits set to one. We first permute the bits of the key according to the following permutation (which takes a 64-bit input and produces a 56-bit output, hence discarding the parity bits).

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

The output of this permutation, called PC-1 in the literature, is divided into a 28-bit left half C_0 and a 28-bit right half D_0. Now for each round we compute

$$C_i \leftarrow C_{i-1} \lll p_i,$$
$$D_i \leftarrow D_{i-1} \lll p_i,$$

where $x \lll p_i$ means perform a cyclic shift on x to the left by p_i positions. If the round number i is $1, 2, 9$ or 16 then we shift left by one position, otherwise we shift left by two positions. Finally the two portions C_i and D_i are joined back together and are subject to another permutation, called PC-2, to produce the final 48-bit round key. The permutation PC-2 is described below.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

13.3. AES

The AES winner was decided in autumn 2000 to be the Rijndael algorithm designed by Joan Daemen and Vincent Rijmen. AES is a block cipher which does not rely on the basic design of the Feistel cipher; instead it is designed as a *substitution-permutation* network, or SP-network. However, AES does have a number of similarities with DES. Block ciphers based on the SP-network design consist

of a series of rounds, each of which consists of a key addition phase, a substitution phase and a permutation phase. The idea is that the permutation phase aims to produce an avalanche effect, by spreading out differences in the input to other parts of the state as quickly as possible, performing a process called *diffusion*. The substitution phase is the main non-linear component and this aims to introduce as much non-linearity, or *confusion*, into the output as possible.

AES has, unlike DES, a strong mathematical structure, as most of its operations are based on arithmetic in the finite fields $\mathbb{F}_{2^8}$ and $\mathbb{F}_2$. However, unlike DES the encryption and decryption operations are distinct, and do not just require a re-ordering of the round keys.

Recall from Chapter 6 that elements of $\mathbb{F}_{2^8}$ are stored as bit vectors (or bytes) representing binary polynomials. For example the byte given by `0x83` in hexadecimal gives the bit pattern

$$1, 0, 0, 0, 0, 0, 1, 1$$

since

$$\texttt{0x83} = 8 \cdot 16 + 3 = 131$$

in decimal. One can obtain the bit pattern directly by noticing that 8 in binary is $1, 0, 0, 0$ and 3 in 4-bit binary is $0, 0, 1, 1$ and one simply concatenates these two bit strings together. The bit pattern itself then corresponds to the binary polynomial

$$x^7 + x + 1.$$

So we say that the hexadecimal number `0x83` represents the binary polynomial

$$x^7 + x + 1.$$

Arithmetic in $\mathbb{F}_{2^8}$ in the AES algorithm is performed using polynomial arithmetic modulo the irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

AES identifies 32-bit words with polynomials in $\mathbb{F}_{2^8}[X]$ of degree less than four. This is done in a big-endian format, in that the smallest index corresponds to the least important coefficient. Hence, the word

$$a_0 \| a_1 \| a_2 \| a_3$$

will correspond to the polynomial

$$a_3 \cdot X^3 + a_2 \cdot X^2 + a_1 \cdot X + a_0.$$

Arithmetic is performed on polynomials in $\mathbb{F}_{2^8}[X]$ modulo the reducible polynomial

$$M(X) = X^4 + 1.$$

Hence, arithmetic is done on these polynomials in a ring rather than a field, since $M(X)$ is reducible.

Rijndael was a parametrized algorithm, in that it could operate on block sizes of 128, 192 or 256 bits, but in the final AES standard the block size was fixed at 128 bits. However, AES does support keys of size 128, 192 or 256 bits. For each key size a different number of rounds is specified. To make our discussion simpler we shall consider the simpler, and probably more used, variant which uses a block size of 128 bits and a key size of 128 bits, in which case 10 rounds are specified. From now on our discussion is only of this simpler version.

AES operates on an internal four-by-four matrix of bytes, called the state matrix

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix},$$

which is usually held as a vector of four 32-bit words, each word representing a column. Each round key is also held as a four-by-four matrix

$$K_i = \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix}.$$

13.3.1. AES Operations: The AES round function operates using a set of four operations which we shall now describe.

SubBytes: Two types of S-Boxes are used in AES: one for the encryption rounds and one for the decryption rounds, each one being the inverse of the other. We shall describe the encryption S-Box; the decryption one will follow immediately. The S-Boxes of DES were chosen by searching through a large space of possible S-Boxes, so as to avoid attacks such as differential cryptanalysis. The S-Box of AES is chosen to have a simple mathematical structure, which allows one to formally argue how resilient the cipher is to differential and linear cryptanalysis. Not only does this mathematical structure help protect against differential cryptanalysis, but it also convinces users that it has not been engineered with some hidden trapdoor.

Each byte $s = [s_7, \ldots, s_0]$ of the AES state matrix is taken in turn and considered as an element of $\mathbb{F}_{2^8}$. The S-Box can be mathematically described in two steps:

(1) The multiplicative inverse of s in $\mathbb{F}_{2^8}$ is computed, to produce a new byte $x = [x_7, \ldots, x_0]$. For the element $[0, \ldots, 0]$, which has no multiplicative inverse, one uses the convention that this is mapped to zero, so as to maintain a one-to-one mapping from the input to the output of the S-Box.

(2) The bitvector x is then mapped, via the following affine $\mathbb{F}_2$ transformation, to the bitvector y:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The new byte is given by y. The decryption S-Box is obtained by first inverting the affine transformation and then taking the multiplicative inverse. These byte substitutions can either be implemented using table look-up or by implementing circuits, or code, which implement the inverse operation in $\mathbb{F}_{2^8}$ and the affine transformation.

ShiftRows: The ShiftRows operation in AES performs a cyclic shift on the state matrix. Each row is shifted by different offsets. For AES this is given by

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \longmapsto \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}.$$

The inverse of the ShiftRows operation is simply the equivalent shift in the opposite direction. The ShiftRows operation ensures that the columns of the state matrix "interact" with each other over a number of rounds.

MixColumns: The MixColumns operation ensures that the rows in the state matrix "interact" with each other over a number of rounds; combined with the ShiftRows operation it ensures each byte of the output state depends on each byte of the input state. We consider each column of the state $[a_0, a_1, a_2, a_3]$ in turn, and consider it as a polynomial of degree less than four with coefficients in $\mathbb{F}_{2^8}$. The new column $[b_0, b_1, b_2, b_3]$ is produced by taking the polynomial

$$a(X) = a_0 + a_1 \cdot X + a_2 \cdot X^2 + a_3 \cdot X^3$$

and multiplying it by the polynomial

$$c(X) = \texttt{0x02} + \texttt{0x01} \cdot X + \texttt{0x01} \cdot X^2 + \texttt{0x03} \cdot X^3$$

modulo

$$M(X) = X^4 + 1.$$

This operation is conveniently represented by the following matrix operation in $\mathbb{F}_{2^8}$,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \leftarrow \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

In $\mathbb{F}_{2^8}$ the above matrix is invertible, hence the inverse of the MixColumns operation can also be implemented using a matrix multiplication such as that above.

AddRoundKey: The round key addition is particularly simple. One takes the state matrix and exclusive-or's it, byte by byte, with the round key matrix. The inverse of this operation is clearly the same operation.

Round Structure: The AES algorithm can now be described using the pseudo-code in Algorithm 13.1. The message block to encrypt is assumed to be entered into the state matrix S, the output encrypted block is also given by the state matrix S. Notice that the final round does not perform a MixColumns operation. The corresponding decryption operation is described in Algorithm 13.2.

Algorithm 13.1: AES encryption outline

```
AddRoundKey(S, K_0).
for i = 1 to 9 do
    SubBytes(S).
    ShiftRows(S).
    MixColumns(S).
    AddRoundKey(S, K_i).
SubBytes(S).
ShiftRows(S).
AddRoundKey(S, K_10).
```

Algorithm 13.2: AES decryption outline

AddRoundKey(S, K_{10}).
InverseShiftRows(S).
InverseSubBytes(S).
for $i = 9$ **downto** 1 **do**
  AddRoundKey(S, K_i).
  InverseMixColumns(S).
  InverseShiftRows(S).
  InverseSubBytes(S).
AddRoundKey(S, K_0).

AES Key Schedule: The only thing left to describe is how AES computes the round keys from the main key. Recall that the main key is 128 bits long, and we need to produce 11 round keys $K_0, \ldots, K_{11}$ all of which consist of four 32-bit words, each word corresponding to a column of a matrix as described above. The key schedule makes use of a round constant which we shall denote by

$$RC_i \leftarrow x^i \pmod{x^8 + x^4 + x^3 + x + 1}.$$

We label the round keys as $(W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3})$ where i is the round. The initial main key is first divided into four 32-bit words (k_0, k_1, k_2, k_3). The round keys are then computed as in Algorithm 13.3, where RotBytes is the function which rotates a word to the left by a single byte, and SubBytes applies the AES encryption S-Box to every byte in a word.

Algorithm 13.3: AES key schedule

$W_0 \leftarrow K_0, W_1 \leftarrow K_1, W_2 \leftarrow K_2, W_3 \leftarrow K_3$.
for $i \leftarrow 1$ **to** 10 **do**
  $T \leftarrow$RotBytes(W_{4i-1}).
  $T \leftarrow$SubBytes(T).
  $T \leftarrow T \oplus RC_i$.
  $W_{4i} \leftarrow W_{4i-4} \oplus T$.
  $W_{4i+1} \leftarrow W_{4i-3} \oplus W_{4i}$.
  $W_{4i+2} \leftarrow W_{4i-2} \oplus W_{4i+1}$.
  $W_{4i+3} \leftarrow W_{4i-1} \oplus W_{4i+2}$.

13.4. Modes of Operation

A block cipher like DES or AES can be used in a variety of ways to encrypt a data string. Soon after DES was standardized another US Federal standard appeared giving four *recommended* ways of using DES for data encryption. These modes of operation have since been standardized internationally and can be used with any block cipher. The four modes are

- **ECB Mode:** This is simple to use, but suffers from possible deletion and insertion attacks. A one-bit error in the ciphertext gives a one whole block error in the decrypted plaintext.
- **CBC Mode:** This is probably the best of the original four modes of operation, since it helps protect against deletion and insertion attacks. In this mode a one-bit error in the ciphertext gives not only a one-block error in the corresponding plaintext but also a one-bit error in the next decrypted plaintext block.

- **OFB Mode:** This mode turns a block cipher into a stream cipher. It has the property that a one-bit error in the ciphertext gives a one-bit error in the decrypted plaintext.
- **CFB Mode:** This mode also turns a block cipher into a stream cipher. A single bit error in the ciphertext affects both this block and the next, just as in CBC mode.

Over the years various other modes of operation have been presented. Probably the most popular of the more modern modes is

- **CTR Mode:** This also turns the block cipher into a stream cipher, but it enables blocks to be processed in parallel, thus providing performance advantages when parallel processing is available.

We shall now describe each of these five modes of operation in detail, and show what security properties each has (or in most cases has not). Finally, we present a summary of these five basic modes of operation in Tables 13.1 and 13.2. Throughout this section we ignore difficulties related to which padding scheme should be used to pad a message out to a multiple of the block length. We discuss padding schemes in Chapter 14. In the following discussion we let $\mathsf{ECB}[F], \mathsf{CBC}[F], \mathsf{OFB}[F], \mathsf{CFB}[F]$ and $\mathsf{CTR}[F]$ denote the mode of operation when instantiated with the function F, which could be a block cipher, a pseudo-random permutation or a pseudo-random function depending on the situation we are considering.

13.4.1. ECB Mode: Electronic Code Book Mode, or ECB Mode, is the simplest way to use a block cipher. The data to be encrypted m is divided into blocks of n bits:

$$m_1, m_2, \ldots, m_q$$

with the last block padded if needed. The ciphertext blocks $c_1, \ldots, c_q$ are then defined as follows

$$c_i \leftarrow e_k(m_i),$$

as described in Figure 13.7. Decipherment is simply the reverse operation as explained in Figure 13.8.

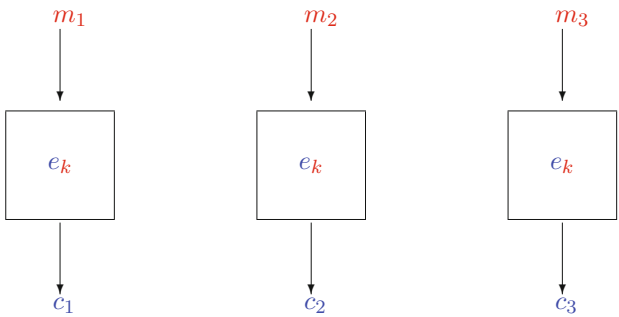


FIGURE 13.7. ECB encipherment

ECB Mode has a number of problems: the first is due to the property that if $m_i = m_j$ then we have $c_i = c_j$, i.e. the same input block always generates the same output block. This is a problem in practice since stereotyped beginnings and endings of messages are common. The second problem comes because we could simply delete blocks from the message and no one would know. Thirdly we could replay known blocks from other messages. By extracting ciphertext corresponding to a known piece of plaintext we can then amend other transactions to contain this known block of text. In terms of our security models from Chapter 11 we have the following.

Theorem 13.2. *ECB Mode* is not IND-PASS *secure.*

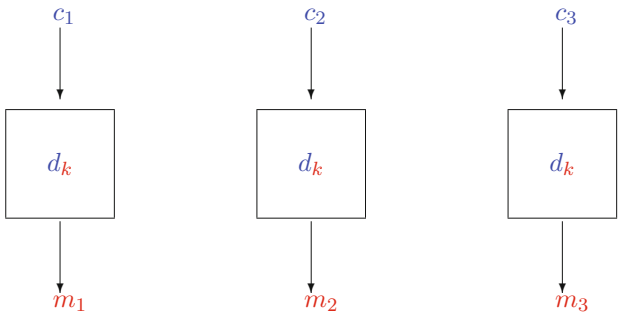


FIGURE 13.8. ECB decipherment

PROOF. To prove something is not secure we need to exhibit an attack within the model. The attack on ECB mode is very simple:

- Let $\mathbf{0}$ denote the block of all zeros, and $\mathbf{1}$ denote the block of all ones.
- Call the $\mathcal{O}_{LR}$ oracle with $m_0 = \mathbf{0}\|\mathbf{1}$ and $m_1 = \mathbf{1}\|\mathbf{1}$. The challenge ciphertext c^* is returned which is the encryption of m_b, for the hidden bit b. The challenge ciphertext consists of two blocks c_0 and c_1
- If $c_0 \neq c_1$ then output $b' = 0$, else return $b' = 1$.

□

It is clear this attack works with a one hundred percent success rate, this is due to the fact that ECB Mode is deterministic. One should also note that this attack is stronger than the standard attack on deterministic encryption schemes (which require access to an encryption oracle). The ECB attack does not require such an oracle. Even if we restrict ourseleves to a notion of one-way security ECB mode is not as secure as we would like

Theorem 13.3. *ECB Mode* is not OW-CCA *secure.*

PROOF. Again, to prove something is not secure we need to exhibit an attack within the model, and again the attack is very simple:

- Let c^* denote the target ciphertext to be decrypted.
- Let r denote an arbitrary block, and form the ciphertext $c \leftarrow c^*\|r$.
- Pass c to the decryption oracle $\mathcal{O}_{d_k}$ to obtain the plaintext $m^*\|s$ for some block s.
- Return m^*.

□

However, we can show the following positive result

Theorem 13.4. *ECB mode* is OW-CPA *secure assuming the underlying block cipher* e_k *acts like a pseudo-random permutation. In particular, let* A *denote an adversary against ECB Mode which makes* q_e *queries to its encryption oracle, where each query is a single block in length, and where the challenge ciphertext is* ℓ distinct *blocks in length*[2]*. There is an adversary* B *such that*

$$\mathrm{Adv}^{\text{OW-CPA}}_{\text{ECB}[e_k]}(A; q_e) \leq \mathrm{Adv}^{\text{PRP}}_{e_k}(B) + \frac{\ell \cdot q_e}{2^n}.$$

where n *is the block size of the cipher* e_k*.*

[2]These assumptions can be changed by making a suitably more complex probability estimate.

PROOF. Our main step in the proof is to replace the underlying block cipher e_k by a pseudo-random permutation. This can be done by the assumption that e_k is a secure PRP, namely there is some adversary B such that

$$\text{Adv}^{\mathsf{PRP}}_{e_k}(B) = \Big|\Pr[A \text{ wins } \mathsf{ECB}[e_k]] - \Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]]\Big| \tag{18}$$

where we let $\mathcal{P}$ denote a random permutation. We now need to bound the probability that A wins this latter game, i.e. $\Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]]$. It is easier to bound the probability that A does not win. Since for a PRP the adversary cannot learn anything about the output value of the permutation until she queries the permutation on the specific input value, the probability that she does not win is given by the probability that out of the q_e distinct queries to the encryption oracle we do not obtain *all* of the ℓ blocks in the challenge ciphertext. Setting $N = 2^n$ this gives us, where ${}^xC_y$ is the function which returns the number of combinations of y objects selected from n,

$$\begin{aligned}
\Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]] &= 1 - \Pr[A \text{ does not win } \mathsf{ECB}[\mathcal{P}]] \\
&\leq 1 - \frac{\ell \cdot {}^{N-1}C_{q_e}}{{}^{N}C_{q_e}} \\
&= 1 - \ell \cdot \left(\frac{(N-1)!}{q_e! \cdot (N-1-q_e)!}\right) \cdot \left(\frac{q_e! \cdot (N-q_e)!}{N!}\right) \\
&= 1 - \ell \cdot \left(\frac{N-q_e}{N}\right) = \frac{N - \ell \cdot (N-q_e)}{N} \\
&\leq \ell \cdot q_e / N.
\end{aligned}$$

Hence,

$$\begin{aligned}
\Pr[A \text{ wins } \mathsf{ECB}[e_k]] &= \Big|\Pr[A \text{ wins } \mathsf{ECB}[e_k]] \\
&\qquad + (\Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]] - \Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]])\Big| && \text{adding in zero} \\
&\leq \Big|\Pr[A \text{ wins } \mathsf{ECB}[e_k]] - \Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]]\Big| \\
&\qquad + \Big|\Pr[A \text{ wins } \mathsf{ECB}[\mathcal{P}]]\Big| && \text{triangle inequality} \\
&\leq \text{Adv}^{\mathsf{PRP}}_{e_k}(B) + \ell \cdot \frac{q_e}{2^n}.
\end{aligned}$$

□

Notice that when q_e is small relative to 2^n the probability $q_e/2^n$ is very close to zero, whereas as q_e approaches 2^n we obtain a probability close to one.

13.4.2. CBC Mode: One way of countering the problems with ECB Mode is to chain the cipher, and in this way add context to each ciphertext block. The easiest way of doing this is to use Cipher Block Chaining Mode, or CBC Mode. Again, the plaintext must first be divided into a series of blocks

$$m_1, \ldots, m_q,$$

and as before the final block may need padding to make the plaintext length a multiple of the block length. Encryption is then performed as in Figure 13.9, or equivalently via the equations

$$\begin{aligned}
c_1 &\leftarrow e_k(m_1 \oplus IV), \\
c_i &\leftarrow e_k(m_i \oplus c_{i-1}) \text{ for } i > 1.
\end{aligned}$$

With the output ciphertext being $IV \| c_1 \| c_2 \| \ldots$. The transmission of IV with the ciphertext can be dropped if the receiver will know what the value will be a priori.

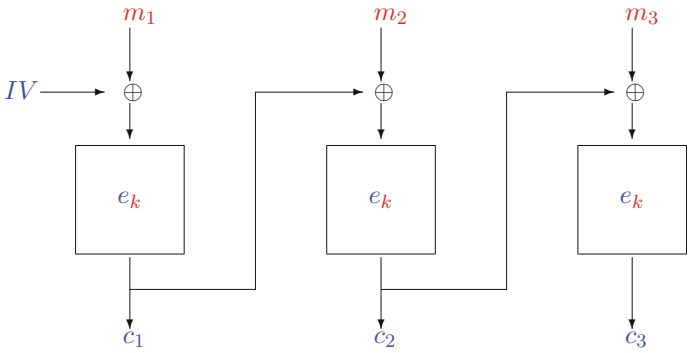


FIGURE 13.9. CBC encipherment

Decryption also requires the IV and is performed as in Figure 13.10, or via the equations

$$m_1 \leftarrow d_k(c_1) \oplus IV,$$
$$m_i \leftarrow d_k(c_i) \oplus c_{i-1} \text{ for } i > 1.$$

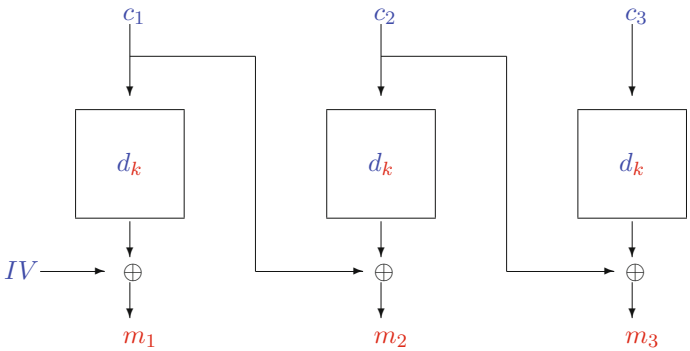


FIGURE 13.10. CBC decipherment

With ECB Mode a single-bit error in transmission of the ciphertext will result in a whole block being decrypted wrongly, whilst in CBC Mode we see that not only will we decrypt a block incorrectly but the error will also affect a single bit of the next block.

Notice that we require an additional initial value IV. This is either a unique, i.e. never repeated, value passed to the encryption function (in which case we are said to have a nonce-based encryption scheme), or it is fixed to a specific value (in which case we have a deterministic variant of CBC Mode), or it is a truly random value chosen internally by the mode of operation. Each of these choices leads to entirely different security properties as we shall now demonstrate. When a non-fixed IV is used the IV value is prepended to the ciphertext.

Fixed IV: Clearly with a fixed IV CBC Mode is deterministic and hence cannot be IND-CPA secure, however one can show it is IND-PASS secure (unlike ECB Mode). This follows from the proof method of IND-CPA security in the random IV case given below. The crucial point to note is that whilst the adversary has control over the input to the first call to the block cipher, he has no control over the other calls when encrypting a multi-block message.

Nonce IV: Here we think of CBC Mode as a nonce-based encryption scheme, as in Section 11.6.4. We start with the negative result

Theorem 13.5. *With a nonce as the IV, CBC Mode* is not IND-CPA *secure.*

PROOF. Let $\mathbf{0}$ be the all-zero block and $\mathbf{1}$ be the all-one block. The attack on the IND-CPA security is as follows:

- Send the message $\mathbf{0}$ with the nonce $IV = \mathbf{0}$ to the encryption oracle $\mathcal{O}_{e_k}$. The adversary obtains the ciphertext $\mathbf{0}\|c$ in return, where $c = e_k(\mathbf{0})$.
- Now send the messages $m_0 = \mathbf{0}$ and $m_1 = \mathbf{1}$ to the $\mathcal{O}_{\mathsf{LR}}$ oracle, with nonce $\mathbf{1}$. Notice this is a new nonce and so the encryption is allowed in the game. Let $\mathbf{1}\|c^*$ be the returned ciphertext.
- If $c^* = c$ then return $b' = 1$, else return $b' = 0$.

To see why this attack works, note that if the hidden bit is $b = 1$ then the challenger returns c^* which is the evaluation of the block cipher on the block $\mathbf{1} \oplus \mathbf{1} = \mathbf{0}$. Whereas if $b = 0$ then the evaluation is on the block $\mathbf{0} \oplus \mathbf{1} = \mathbf{1}$. □

On the positive side, when used only once nonce-based encryption is identical to a fixed IV, and so CBC Mode used in a nonce-based encryption methodology is IND-PASS secure.

Random IV: With a random IV we can be more positive, since CBC Mode is IND-CPA secure as we will now show.

Theorem 13.6. *With a random IV, CBC Mode is* IND-CPA *secure assuming the underlying block cipher e_k acts like a pseudo-random permutation. In particular let A denote an adversary against CBC Mode which makes q_e queries to its encryption oracle, and let all plaintext submitted to both the* LR *and encryption oracles be at most ℓ blocks in length. Then there is an adversary B such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CBC}[e_k]}(A; q_e) \leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{3 \cdot T^2}{2^n},$$

where n is the block size of the cipher e_k and $T = (q_e + 1) \cdot \ell$.

PROOF. In the security game the challenger needs to call the underlying block cipher on behalf of the adversary. The total number of such calls is bounded by $T = (q_e + 1) \cdot \ell$.

Our first step in the proof is to replace the underlying block cipher e_k by a pseudo-random permutation. This can be done by the assumption that e_k is a secure PRP, namely there is some adversary B such that

$$\mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) = \Big|\Pr[A \text{ wins } \mathsf{CBC}[e_k]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]]\Big| \tag{19}$$

where we let $\mathcal{P}$ denote a random permutation. Our next step is to switch from the component being a random *permutation* to a random *function*. This follows in the same way as we proved the PRF-PRP Switching Lemma (Lemma 11.2). Suppose we replace $\mathcal{P}$ by a random function $\mathcal{F}$ in the CBC game and we let E denote the event, during the game $\mathsf{CBC}[\mathcal{F}]$, that the adversary makes two

calls to $\mathcal{F}$ which result in the same output value. We have

$$\begin{aligned}\left|\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]]\right| &= \Big|\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] \\ &\qquad - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \wedge \neg E] \\ &\qquad - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \wedge E]\Big| \\ &= \Big|\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] \\ &\qquad - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \mid E] \cdot \Pr[E]\Big| \\ &\leq \Pr[E] \leq \frac{T^2}{2^{n+1}}, \end{aligned} \tag{20}$$

since if E does not happen the two games are identical from the point of view of the adversary, and by the birthday bound $\Pr[E] \leq \frac{T^2}{2^{n+1}}$.

Our final task is to bound the probability of A winning the CBC game when the underlying "block cipher" is a random function. First let us consider how the challenger works in the game $\mathsf{CBC}[\mathcal{F}]$. When the adversary makes an $\mathcal{O}_{\mathsf{LR}}$ or $\mathcal{O}_{e_k}$ call, the challenger answers the query by calling the random function. As we are dealing with a random function, and not a random permutation, the challenger can select the output value of $\mathcal{F}$ *independently* from the codomain; i.e. it does not need to adjust the output values depending on the previous values. This last point will make our analysis simpler, and is why we switched to the PRF game from the PRP game.

Now notice that the adversary does not control the inputs to the random function at any stage in the game, so the only way he can find any information is by creating an input collision, i.e. two calls the challenger makes to the random function are on the same input values[3].

We thus let M_j denote the event that the adversary makes an input collision happen within the first j calls, and note that if M_T does not happen then the adversary's probability of winning is $1/2$, i.e. the best he can do is guess. We have

$$\begin{aligned}\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]] &= \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \mid M_T] \cdot \Pr[M_T] \\ &\qquad + \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \mid \neg M_T] \cdot \Pr[\neg M_T] \\ &\leq \Pr[M_T] + \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}] \mid \neg M_T] \\ &\leq \Pr[M_T] + \frac{1}{2} \end{aligned} \tag{21}$$

So we are left with estimating $\Pr[M_T]$. Clearly, we have $\Pr[M_1] = 0$, and for all $j \geq 1$ we have

$$\Pr[M_j] \leq \Pr[M_{j-1}] + \Pr[M_j \mid \neg M_{j-1}],$$

from which it follows that

$$\Pr[M_j] \leq \frac{(j-1) \cdot (q_e + 1) \cdot \ell}{2^n}.$$

From which follows $\Pr[M_T] \leq T^2/2^n$.

[3]This is why CBC Mode is not secure in the nonce-based setting as in this setting the adversary controls the first input block, by selecting the first block of the message and the IV.

Summing up we have

$$\begin{aligned}
\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CBC}[e_k]}(A; q_e) &= 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[e_k]] - \frac{1}{2} \right| \\
&= 2 \cdot \Big| \Pr[A \text{ wins } \mathsf{CBC}[e_k]] - \frac{1}{2} \\
&\qquad + (\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]]) \Big| && \text{adding in zero} \\
&\leq 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[e_k]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] \right| \\
&\qquad + 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \frac{1}{2} \right| && \text{triangle inequality} \\
&= \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \frac{1}{2} \right| && \text{by equation (19)} \\
&= \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + 2 \cdot \Big| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \frac{1}{2} \\
&\qquad + (\Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]]) \Big| && \text{adding in zero} \\
&\leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) \\
&\qquad + 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{P}]] - \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]] \right| \\
&\qquad + 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]] - \frac{1}{2} \right| && \text{triangle inequality} \\
&\leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{T^2}{2^n} + 2 \cdot \left| \Pr[A \text{ wins } \mathsf{CBC}[\mathcal{F}]] - \frac{1}{2} \right| && \text{by equation (20)} \\
&\leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{T^2}{2^n} + 2 \cdot \Pr[M_T] && \text{by equation (21)} \\
&\leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{3 \cdot T^2}{2^n}.
\end{aligned}$$

□

Let us examine what this means when we use the AES block cipher in CBC Mode. First the block length of AES is $n = 128$, and let us assume the key size is 128 as well. If we assume AES behaves as a PRP, then we expect that

$$\mathrm{Adv}^{\mathsf{PRP}}_{AES}(B) \leq \frac{1}{2^{128}}$$

for all adversaries B. We can now work out the advantage *for any* adversary A to break AES when used in CBC mode, in the sense of IND-CPA. We find

$$\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CBC}[AES]}(A; q_e) \leq \frac{1 + 3 \cdot T^2}{2^{128}}.$$

Thus even if the adversary makes 2^{30} calls to the underlying block cipher, the advantage will still be less than

$$\frac{1 + 3 \cdot 2^{60}}{2^{128}} \approx 2^{-66},$$

which is incredibly small. Thus as long as we restrict the usage of AES in CBC Mode with a random IV to encrypting around 2^{30} blocks per key we will have a secure cipher. Restricting the usage of a symmetric cipher per key is enabled by requiring a user to generate a new key every so often.

For all three variants CBC Mode is not OW-CCA secure and hence not IND-CCA secure, due to a similar attack to that in Theorem 13.3. The OW-CPA security of CBC Mode, for all three ways of picking the IV, follows from Theorem 13.4 due to the proof of the following theorem.

Theorem 13.7. *For any method of choosing the IV, CBC mode* is OW-CPA *secure assuming the underlying block cipher e_k acts like a pseudo-random permutation. In particular let A denote an adversary against CBC Mode which makes q_e queries to its encryption oracle (with each query being at most ℓ blocks in length), then there is an adversary B such that*

$$\mathrm{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{CBC}[e_k]}(A; q_e) = \mathrm{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{ECB}[e_k]}(B; q_e \cdot \ell).$$

PROOF. For uniform challenge messages the distribution (for a fixed number of blocks) of the challenges given to adversary A and adversary B are identical. Algorithm B works as follows. We let $c^* = c_1 \| c_2 \| \dots$ denote the challenge ciphertext passed to adversary B; we pass this to adversary A who returns the CBC Mode decryption of c^*, with the IV being anything that the given method allows, we let $m'_1 \| m'_2 \| \dots$ be the returned plaintext. When algorithm A makes an encryption oracle request; this is answered by calling algorithm B's encryption oracle a block at a time, and simulating CBC Mode. Then using the known IV used when passing c^* to A, algorithm B can decrypt c^* under ECB Mode to obtain $m_1 \| m_2 \| \dots$, using the following equations:

$$m_1 = m'_1 \oplus IV,$$
$$m_i = m'_i \oplus c_i \text{ for } i > 1.$$

□

13.4.3. OFB Mode: Output Feedback Mode, or OFB Mode, enables a block cipher to be used as a stream cipher. We use the block cipher to create the keystream, n bits at a time, where n is the block size. Again we divide the plaintext into a series of blocks, each block being n-bits long:

$$m_1, \dots, m_q.$$

Encryption is performed as follows; see Figure 13.11 for a graphical representation. First we set $Y_0 \leftarrow IV$, then for $i = 1, 2, \dots, q$, we perform the following steps,

$$Y_i \leftarrow e_k(Y_{i-1}),$$
$$c_i \leftarrow m_i \oplus Y_i.$$

The output ciphertext is $IV \| c_1 \| c_2 \| \dots$. Decipherment in OFB Mode is performed in a similar manner.

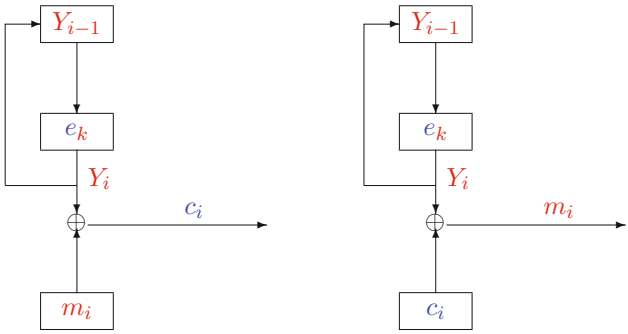


FIGURE 13.11. OFB encipherment and decipherment

We now turn to discussing security of OFB Mode. Clearly, when used with a fixed IV, OFB Mode is *not* IND-CPA secure; it turns out it is not OW-CPA secure either when used with a fixed IV as the next result demonstrates.

Theorem 13.8. *When used with a fixed IV, OFB Mode is not* OW-CPA *secure.*

PROOF. Call the encryption oracle on the plaintext consisting of all zero blocks. This reveals the "keystream", which can then be exclusive-or-ed with the challenge ciphertext to produce the required plaintext. □

Also on the negative side, for any method of selecting the IV, OFB Mode *is not* OW-CCA secure and hence is not IND-CCA secure; again the "attack" is exactly the same as in Theorem 13.3. On another negative side we have the following.

Theorem 13.9. *OFB Mode is not OW-CPA secure in the nonce-based setting, i.e. when the* IV *is a nonce.*

PROOF. The adversary first picks a nonce n and asks the encryption oracle for an encryption of $m' = m'_1 \| m'_2 \| \ldots$. The ciphertext $c_1 \| c_2 \| \ldots$ will be returned, from which the adversary can work out the keystream starting from IV n, and hence also from IV, $c_1 \oplus m'_1 = e_k(n)$. The adversary now asks for the challenge ciphertext c^* with nonce $c_1 \oplus m'_1 = e_k(n)$. Using the previously obtained keystream the adversary can recover the message encrypted by c^*. □

So we are left to consider what positive results hold. It also turns out that the proof of Theorem 13.6 can be applied to OFB Mode as well as CBC Mode, so we have that OFB Mode is IND-CPA secure, assuming the underlying block cipher is a secure pseudo-random permutation, when used with a random IV.

13.4.4. CFB Mode: The next mode we consider is called Cipher FeedBack Mode, or CFB Mode. This is very similar to OFB Mode in that we use the block cipher to produce a stream cipher. Recall that in OFB Mode the keystream was generated by encrypting IV and then iteratively encrypting the output from the previous encryption. In CFB Mode the keystream output is produced by the encryption of the ciphertext, as in Figure 13.12, by the following steps, upon setting $Y_0 \leftarrow IV$,

$$Z_i \leftarrow e_k(Y_{i-1}),$$
$$Y_i \leftarrow m_i \oplus Z_i.$$

We do not present the decryption steps, but leave this as an exercise for the reader. CFB mode has

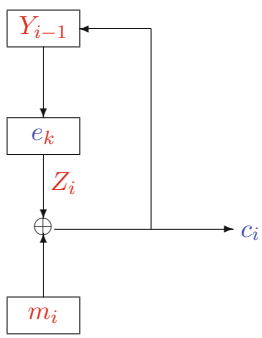


FIGURE 13.12. CFB encipherment

an interesting attack against it. Suppose the challenge ciphertext is $IV \| c_1 \| c_2 \| \ldots$, then if we query the *encryption* oracle with the zero plaintext block, but IV equal to c_1, we will obtain a ciphertext

c' such that $c_2 \oplus c' = m_2$. Continuing in this way we can recover all the plaintext blocks bar the first one. This clearly leads to an attack against IND security for nonce-based encryption. However it does not lead to an attack against the full one-wayness, as one cannot recover the first block.

Theorem 13.10. *CFB Mode* is not IND-CPA *secure when used as a nonce-based encryption scheme.*

Just as with OFB Mode, the proof of Theorem 13.6 can be applied, so we have that CFB Mode with a random IV is IND-CPA secure assuming the underlying block cipher is a secure pseudo-random permutation. We can also use the proof of Theorem 13.4, to show that CFB mode is OW-CPA secure when used as a nonce-based/random IV scheme.

13.4.5. CTR Mode: The next mode we consider is called Counter Mode, or CTR Mode. This combines many of the advantages of ECB Mode, but with none of the disadvantages. We first select a public IV, or counter, which is chosen differently for each message encrypted under the fixed key k. Then encryption proceeds for the ith block, by encrypting the value of $IV + i$ and then exclusive-or'ing this with the message block. In other words we have

$$c_i \leftarrow m_i \oplus e_k(IV \oplus \langle i \rangle_n),$$

where $\langle i \rangle_n$ is the n-bit representation of the number i. This is explained pictorially in Figure 13.13. An important property to preserve security of CTR Mode is that the counter input to *any* of the block cipher calls may not be reused in any subsequent encryption. In the case of a random IV chosen by the encryptor this will happen with overwhelming probability, assuming we limit the number of block cipher invocations with a given key. In the case of nonce-based encryption we simply have to ensure that if we encrypt a t-block message with nonce $IV = j$, then we never take a new nonce in the range $[j, \ldots, j + t - 1]$.

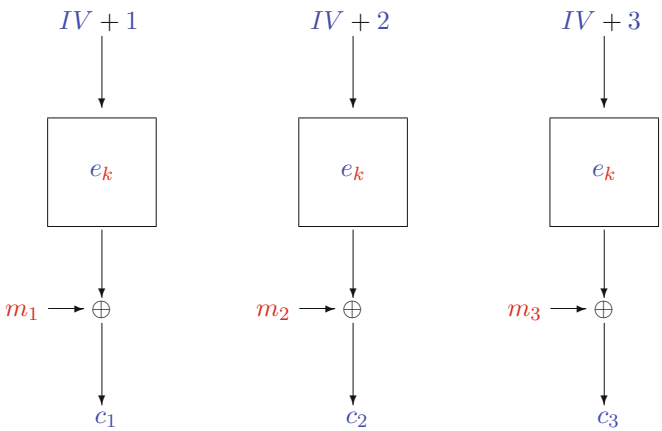


FIGURE 13.13. CTR encipherment

CTR Mode has a number of interesting properties. Firstly, since each block can be encrypted independently, much like in ECB Mode, we can process each block at the same time. Compare this to CBC Mode, OFB Mode or CFB Mode where we cannot start encrypting the second block until the first block has been encrypted. This means that encryption, and decryption, can be performed in parallel. Another performance advantage comes from the fact that we only ever apply the encryption operation of the underlying block cipher.

However, unlike ECB Mode, two equal blocks will not encrypt to the same ciphertext value. This is because each plaintext block is encrypted using a different input to the encryption function;

in some sense we are using the block cipher encryption of the different inputs to produce a stream cipher. Also unlike ECB Mode each ciphertext block corresponds to a precise position within the ciphertext, as one needs the position information to be able to decrypt it successfully.

In terms of security properties, the security proof for IND-CPA security is actually simpler than that used for CBC Mode. The reason is that with the above restrictions on the reuse of IVs we never have to worry about an input collision for the block cipher. Thus, assuming the total number of blocks encrypted with a fixed key is kept relatively low, we have the following.

Theorem 13.11. *CTR Mode is* IND-CPA *secure assuming the underlying block cipher e_k acts like a pseudo-random permutation. In particular let A denote an adversary against CTR Mode which makes q_e queries to its encryption oracle, and let all plaintext submitted to both the* LR *and encryption oracles be at most ℓ blocks in length. There is then an adversary B such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CTR}[e_k]}(A; q_e) \leq \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{T^2}{2^n},$$

where n is the block size of the cipher e_k and $T = (q_e + 1) \cdot \ell$.

With the above restrictions on the reuse of the nonce, we obtain an IND-CPA secure nonce-based encryption scheme with exactly the same advantage statement.

All is not totally positive however; the attack in Theorem 13.8 also applies to CTR mode and hence the scheme is not OW-CPA secure when used with a fixed IV. In addition we cannot achieve CCA security, again due to the attack presented in Theorem 13.3.

Mode	IV	OW-PASS	OW-CPA	OW-CCA
ECB Mode	-	OW-CPA $\implies$ ✓	Thm 13.4 $\implies$ ✓	Thm 13.3 $\implies$ ✗
CBC Mode	fixed	IND-PASS $\implies$ ✓	Thm 13.7 $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
	nonce	IND-PASS $\implies$ ✓	Thm 13.7 $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
	random	IND-PASS $\implies$ ✓	IND-CPA $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
OFB Mode	fixed	IND-PASS $\implies$ ✓	Thm 13.8 $\implies$ ✗	Prf of Thm 13.3 $\implies$ ✗
	nonce	IND-PASS $\implies$ ✓	Thm 13.9 $\implies$ ✗	Prf of Thm 13.3 $\implies$ ✗
	random	IND-PASS $\implies$ ✓	IND-CPA $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
CFB Mode	fixed	IND-PASS $\implies$ ✓	(nonce OW-CPA) $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
	nonce	IND-PASS $\implies$ ✓	Prf of Thm 13.4 $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
	random	IND-PASS $\implies$ ✓	IND-CPA $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
CTR Mode	fixed	IND-PASS $\implies$ ✓	Prf of Thm 13.8 $\implies$ ✗	Prf of Thm 13.3 $\implies$ ✗
	nonce	IND-PASS $\implies$ ✓	IND-CPA $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗
	random	IND-PASS $\implies$ ✓	IND-CPA $\implies$ ✓	Prf of Thm 13.3 $\implies$ ✗

TABLE 13.1. One-way security properties of the five basic modes of operation

We summarize our results on modes of operation in Tables 13.1 and 13.2. Against each tick or cross we give the theorem number which presents this result, or the proof of the theorem which can be modified to give the result. For nonce-based CTR mode we assume the convention with respect to nonces described earlier. To derive most of the tables we note the implications given in Figure 11.18; e.g. (¬OW-XXX) $\implies$ (¬IND-XXX) and (equivalently) (IND-XXX) $\implies$ (OW-XXX), also (IND-CPA) $\implies$ (IND-PASS). We also note that in the passive security setting there is no difference between the nonce-based and fixed IV variants, and if a scheme is IND-PASS secure in the random-IV setting, then it is also IND-PASS secure in the nonce-based setting.

Mode	IV	IND-PASS	IND-CPA	IND-CCA
ECB Mode	-	Thm 13.2 $\implies$ ✗	($\neg$ IND-PASS) $\implies$ ✗	($\neg$ IND-PASS) $\implies$ ✗
CBC Mode	fixed	(nonce IND-PASS) $\implies$ ✓	Trivially ✗	Trivially ✗
	nonce	(random IND-PASS) $\implies$ ✓	Thm 13.5 $\implies$ ✗	($\neg$ OW-CCA) $\implies$ ✗
	random	IND-CPA $\implies$ ✓	Thm 13.6 $\implies$ ✓	($\neg$ OW-CCA) $\implies$ ✗
OFB Mode	fixed	(nonce IND-PASS) $\implies$ ✓	Trivially ✗	Trivially ✗
	nonce	(random IND-PASS) $\implies$ ✓	($\neg$ OW-CPA) $\implies$ ✗	($\neg$ OW-CCA) $\implies$ ✗
	random	IND-CPA $\implies$ ✓	Prf of Thm 13.6 $\implies$ ✓	($\neg$ OW-CCA) $\implies$ ✗
CFB Mode	fixed	(nonce IND-PASS) $\implies$ ✓	Trivially ✗	Trivially ✗
	nonce	(random IND-PASS) $\implies$ ✓	Thm 13.10 $\implies$ ✗	($\neg$ OW-CCA) $\implies$ ✗
	random	IND-CPA $\implies$ ✓	Prf of Thm 13.6 $\implies$ ✓	($\neg$ OW-CCA) $\implies$ ✗
CTR Mode	fixed	(nonce IND-PASS) $\implies$ ✓	Trivially ✗	Trivially ✗
	nonce	IND-CPA $\implies$ ✓	Thm 13.11 $\implies$ ✓	($\neg$ OW-CCA) $\implies$ ✗
	random	IND-CPA $\implies$ ✓	Thm 13.11 $\implies$ ✓	($\neg$ OW-CCA) $\implies$ ✗

TABLE 13.2. Indistinguishability security properties of the five basic modes of operation

13.5. Obtaining Chosen Ciphertext Security

All the prior modes of operation did not provide security against chosen ciphertext attacks. To do this one needs more advanced modes of operation, called authenticated encryption modes. There are a number of modes which provide this property for symmetric encryption based on block ciphers, for example CCM Mode, GCM Mode and OCB Mode. There is little room in this book to cover these modes, however we will present a simple technique to obtain an IND-CCA secure symmetric cipher from one of the previous IND-CPA secure modes of operation, called *Encrypt-then-MAC*.

Let (E_k, D_k) denote an IND-CPA symmetric encryption scheme, say CBC Mode or CTR Mode instantiated with AES. Let $\mathbb{K}_1$ denote the key space of the encryption scheme. The problem with the previous chosen ciphertext attacks was that an adversary could create a new ciphertext without needing to know the underlying key. So the decryption oracle would decrypt any old garbage which the adversary threw at it. The trick to obtaining CCA secure schemes is to ensure that the decryption oracle rejects almost all of the ciphertexts that the adversary creates; in fact we hope it rejects all bar the ones validly produced by an encryption oracle.

13.5.1. Encrypt-then-MAC: The standard *generic* method of adding this form of integrity protection to the ciphertext is to append a message authentication code to the ciphertext. We let $(\mathsf{Mac}_k, \mathsf{Verify}_k)$ denote a message authentication code with key space $\mathbb{K}_2$. In the next chapter we will see how such codes can be constructed, but for now just assume we can create one which satisfies the security definition of strong existential unforgeability under chosen message attacks, from Chapter 11. We then construct our CCA secure symmetric encryption scheme, called Encrypt-then-MAC (ETM), as follows:

KeyGen(): Sample $k_1 \leftarrow \mathbb{K}_1$ and $k_2 \leftarrow \mathbb{K}_2$, return $k = (k_1, k_2)$.
$\mathrm{Enc}_k(m)$: Compute the ciphertext $c_1 \leftarrow E_{k_1}(m)$, then form a MAC on the ciphertext from $c_2 \leftarrow \mathsf{Mac}_{k_2}(c_1)$. Return the ciphertext $c = (c_1, c_2)$.
$\mathrm{Dec}_k(c)$: Verify the MAC and decrypt the ciphertext: $v \leftarrow \mathsf{Verify}_{k_2}(c_2, c_1)$, and $m \leftarrow D_{k_1}(c_1)$. If $v = \texttt{valid}$ then return m, else return $\perp$.

Note that the message authentication code is applied to the ciphertext *and not* the plaintext; this is very important to maintain security. Also note that, when decrypting, we decrypt the first ciphertext component c_1, irrespective of whether the MAC in c_2 verifies or not; this is to avoid subtle timing attacks. We now show that this construction is IND-CCA secure, assuming the underlying encryption scheme and message authentication code are suitably secure.

Theorem 13.12. *Let A denote an adversary against the* ETM *scheme constructed from the encryption scheme $\Pi_1 = (\mathrm{Enc}, \mathrm{Dec})$ and the message authentication code $\Pi_2 = (M, V)$, then there are two adversaries B_1 and B_2 such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{ETM}}(A) \leq \mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi_1}(B_1) + \mathrm{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\Pi_2}(B_2).$$

Proof. Let E denote the event that the adversary produces a ciphertext, which is not the output of a call to the encryption oracle, which when passed to the decryption oracle results in the output of something other than $\perp$. We have

$$\Pr[A \text{ wins}] = \Pr[A \text{ wins} \wedge \neg E] \cdot \Pr[\neg E] + \Pr[A \text{ wins} \wedge E] \cdot \Pr[E] \quad \leq \Pr[A \text{ wins} \wedge \neg E] + \Pr[E].$$

Now $\Pr[A \text{ wins} \wedge \neg E]$ is the same probability as running A in a IND-CPA attack, since if E does not happen then A does not learn anything from its decryption queries, and all decryption oracle queries can be answered by either returning $\perp$, or remembering what was queried to the encryption oracle. Hence in this case we can take A to be the B_1 in the theorem, and just ignore any decryption queries which the adversary A makes which do not correspond to the adversary's outputs from the encryption oracle.

If event E happens then the adversary A must have created a ciphertext, which is different from the challenge ciphertext, whose MAC verifies. In such a situation we can create an adversary B_2 which breaks the message authentication code as follows. We create B_2 by using A as a subroutine.

- Generate $k_1 \leftarrow \mathbb{K}_1$.
- Call A.
- When A makes a query to the encryption oracle, use the key k_1 to create a first ciphertext component, and then use B_2's oracle $\mathcal{O}_{\mathsf{Mac}_k}$ to create the second ciphertext component.
- When A makes a query to the $\mathcal{O}_{\mathsf{LR}}$ oracle, pick the random bit b and proceed as for the $\mathcal{O}_{e_k}$ oracle.
- When A makes a query to its decryption oracle we first check whether the ciphertext verifies (using the verification oracle provided to B_2). If it does, and the ciphertext is not the output of $\mathcal{O}_{e_k}$ then stop and return the input as a MAC forgery. Otherwise proceed as in the real game.

Let (c_1, c_2) denote the output of adversary B_2. We note that one of c_1 and c_2 must be different from the output of $\mathcal{O}_{e_k}$ (by definition of B_2) and $\mathcal{O}_{\mathsf{LR}}$ (by the rules A is following). If the difference is c_1 then c_2 is a valid MAC on a message which has not been queried to B_2's $\mathcal{O}_{\mathsf{Mac}_k}$ oracle. Whereas if c_1 is the same as a previous output from $\mathcal{O}_{e_k}$ or $\mathcal{O}_{\mathbb{K}}$, and c_2 is different, then B_2 has managed to produce a *strong* forgery. Thus in either case a MAC forgery has been created and the result follows. So $\Pr[E]$ is bound by the advantage of B_2 in winning the forgery game for the MAC function. □

Note that the same construction can be used to construct a CCA-secure DEM, i.e. a data encapsulation mechanism. Recall that this is a symmetric encryption scheme for which only one message is ever created with a given key, but for which the adversary has a decryption oracle. Using the same technique as in the proof above we can prove the following.

Theorem 13.13. *Let A denote an adversary against the* ETM *scheme constructed from the encryption scheme $\Pi_1 = (\mathrm{Enc}, \mathrm{Dec})$ and the message authentication code $\Pi_2 = (M, V)$, then there are two adversaries B_1 and B_2 such that*

$$\mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\mathsf{ETM}}(A) \leq \mathrm{Adv}^{\mathsf{IND\text{-}PASS}}_{\Pi_1}(B_1) + \mathrm{Adv}^{\mathsf{ot\text{-}sEUF\text{-}CMA}}_{\Pi_2}(B_2).$$

Note that we only require a passively secure encryption scheme and a one-time secure message authentication code. Hence we could, to construct a DEM, use CBC Mode with a fixed IV. This means that the ciphertext for a DEM can be one block shorter than for a general encryption scheme, as we no longer need to transmit the IV. The use of DEMs will become clearer when we discuss hybrid encryption in a later chapter.

13.5.2. Encrypt-and-MAC: We stressed above that it is important that one authenticates the ciphertext and not the message. One popular method in the past for trying to produce a CCA secure symmetric encryption scheme was to use a method called *Encrypt-and-MAC*. Here one applies a MAC to the plaintext and then appends this MAC to the ciphertext. Thus we have

KeyGen(): Sample $k_1 \leftarrow \mathbb{K}_1$ and $k_2 \leftarrow \mathbb{K}_2$, return $k = (k_1, k_2)$.
$\mathrm{Enc}_k(m)$: Compute the ciphertext $c_1 \leftarrow E_{k_1}(m)$, then form a MAC on the plaintext from $c_2 \leftarrow \mathsf{Mac}_{k_2}(m)$. Return the ciphertext $c = (c_1, c_2)$.
$\mathrm{Dec}_k(c)$: Decrypt the ciphertext and then verify the MAC: $m \leftarrow D_{k_1}(c_1)$. $v \leftarrow \mathsf{Verify}_{k_2}(c_2, m)$, If $v = \texttt{valid}$ then return m, else return $\perp$.

The problem is that this method is not *generically* secure. By this we mean that its security depends on the precise choice of the IND-CPA encryption scheme and MAC which one selects. In particular the scheme is *not* secure when instantiated with any of the standard MAC functions we will discuss in Chapter 14, as the following result shows.

Theorem 13.14. *Encrypt-and-MAC is* not IND-CPA *secure when instantiated with an* IND-CPA *encryption scheme and a* deterministic *MAC.*

Proof. The attack is to pick two plaintext messages m_0 and m_1 of the same length and pass these to the $\mathcal{O}_{\mathsf{LR}}$ oracle to obtain a challenge ciphertext $c^* = (c_1^*, c_2^*)$. Now the adversary passes m_0 to its encryption oracle to obtain a new ciphertext $c = (c_1, c_2)$. As the MAC is deterministic, if $c_2 = c_2^*$ then the hidden bit is zero, and if not the hidden bit is one. □

Notice that Encrypt-and-MAC is less secure (in the sense of the IND-CPA notion) than the original encryption algorithm without the MAC!

13.5.3. MAC-then-Encrypt: Another method which has been used in protocols in the past, but which again is not secure in general is called *MAC-then-Encrypt*. Here we MAC the plaintext and then encrypt the plaintext and the MAC together. Thus we have

KeyGen(): Sample $k_1 \leftarrow \mathbb{K}_1$ and $k_2 \leftarrow \mathbb{K}_2$, return $k = (k_1, k_2)$.
$\mathrm{Enc}_k(m)$: Form a MAC on the plaintext from $t \leftarrow \mathsf{Mac}_{k_2}(m)$. Compute the ciphertext $c \leftarrow E_{k_1}(m \| t)$.
$\mathrm{Dec}_k(c)$: Decrypt the ciphertext and then verify the MAC: $m \| t \leftarrow D_{k_1}(c_1)$. $v \leftarrow \mathsf{Verify}_{k_2}(t, m)$. If $v = \texttt{valid}$ then return m, else return $\perp$.

Again this method is not *generically* secure, since we have the following.

Theorem 13.15. *Encryption via the MAC-then-Encrypt method* may not *be* IND-CPA *secure when instantiated with an* IND-CPA *encryption scheme and* EUF-CMA *secure MAC.*

Proof. We present an, admittedly contrived, example although more complex real-life examples do exist. Take an IND-CPA encryption scheme (E_k, D_k) and modify it to form the following encryption scheme, which we shall denote by (E'_k, D'_k). To encrypt one performs outputs $E_k(m) \| 0$, i.e. one adds a zero bit onto the ciphertext output by E_k. To decrypt one ignores the zero bit at the end,

one does not even bother to check it is zero, and decrypts the main component using D_k. It is clear that, since (E_k, D_k) is IND-CPA secure, so is (E'_k, D'_k).

Now form the MAC-then-Encrypt cipher using (E'_k, D'_k) and any EUF-CMA secure MAC. Consider a challenge ciphertext c^*; this will end in zero. Now we can change this zero to a one, to create a new ciphertext c, which is different from c^*. Now since the decryption algorithm does not check the last bit, the decryption oracle will decrypt c to reveal the message underlying c^* and the associated MAC will verify. Thus this MAC-then-Encrypt scheme is not even OW-CCA secure. □

Chapter Summary

- Probably the most famous block cipher is DES, which is itself based on a general design called a Feistel cipher.
- A comparatively recent block cipher is the AES cipher, called Rijndael.
- Both DES and AES obtain their security by repeated application of simple rounds consisting of substitution, permutation and key addition.
- To use a block cipher one needs to also specify a mode of operation. The simplest mode is ECB mode, which has a number of problems associated with it. Hence, it is common to use a more advanced mode such as CBC or CTR mode.
- Some block cipher modes, such as CFB, OFB and CTR modes, allow the block cipher to be used as a stream cipher.
- To obtain an IND-CCA secure scheme one can use Encrypt-then-MAC.

Further Reading

The Rijndael algorithm, the AES process and a detailed discussion of attacks on block ciphers and Rijndael in particular can be found in the book by Daemen and Rijmen. Stinson's book is the best book to explain differential cryptanalysis for students. For a discussion of how to combine encryption functions and MAC functions to obtain IND-CCA secure encryption see the paper by Bellare and Namprempre.

M. Bellare and C. Namprempre. *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm.* Advances in Cryptology – Asiacrypt 2000, LNCS 1976, 531–545, Springer, 2000.

J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard.* Springer, 2002.

D. Stinson. *Cryptography Theory and Practice.* Third Edition. CRC Press, 2005.

CHAPTER 14

Hash Functions, Message Authentication Codes and Key Derivation Functions

Chapter Goals

- To understand the properties of keyed and unkeyed cryptographic hash functions.
- To understand how existing deployed hash functions work.
- To examine the workings of message authentication codes.
- To examine how key derivation functions are constructed from hash functions and message authentication codes.

14.1. Collision Resistance

A cryptographic hash function H is a function which takes arbitrary length bit strings as input and produces a fixed-length bit string as output; the output is often called a digest, hashcode or hash value. Hash functions are used a lot in computer science, but the crucial difference between a standard hash function and a cryptographic hash function is that a cryptographic hash function should at least have the property of being one-way. In other words given any string y from the codomain of H, it should be computationally infeasible to find any value x in the domain of H such that

$$H(x) = y.$$

Another way to describe a hash function which has the one-way property is that it is preimage resistant. Given a hash function which produces outputs of t bits, we would like a function for which finding preimages requires $O(2^t)$ time. Thus the one-way property should match our one-way function security game in Figure 11.5.

A cryptographic hash function should also be second preimage resistant. This is the property that given m it should be hard to find an $m' \neq m$ with $H(m') = H(m)$. The security game for second preimage resistance is given in Figure 14.1. In particular a cryptographic hash function with t-bit outputs should require about 2^t queries before one can find a second preimage. Thus we define the advantage of an adversary to break second preimage resistance of a function H to be

$$\mathrm{Adv}_H^{\mathsf{2nd\text{-}Preimage}}(A) = \Pr[A \text{ wins the } \mathsf{2nd\text{-}Preimage} \text{ game}].$$

We say a function H is second preimage resistant if the advantage is "small" (i.e. about $1/2^t$) for all adversaries A.

In practice we need in addition a property called collision resistance. This is a much harder property to define, and as such we give three definitions, all of which are used in this book, and in practice. First consider a function H mapping elements in a domain D to a codomain C. For collision resistance to make any sense we assume that the domain D is *much larger* than the codomain C. In particular D could be the set of arbitrary length bit strings $\{0,1\}^*$. A function is called collision resistant if it is infeasible to find two distinct values m and m' such that

$$H(m) = H(m').$$

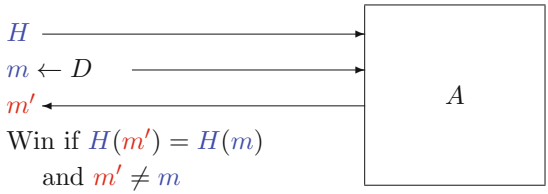


FIGURE 14.1. Security game for second preimage resistance

Pictorially this is described in Figure 14.2.

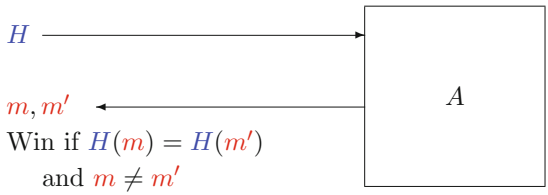


FIGURE 14.2. Security game for collision resistance of a function

The problem with this definition is that we cannot define security. Recall that we say something is secure if the probability of winning the security game is "small", for *all possible* adversaries A. However, there is a trivial adversary which breaks the above game for a *given pre-specified* function H, namely

- Return m and m' such that $H(m) = H(m')$.

Since D is much bigger than C we know such a pair (m, m') must exist, and so must the above adversary.

This looks exactly like the issue we covered when we discussed pseudo-random functions in Chapter 11, and indeed it is. There we got around this problem by assuming the function was taken from a family of functions, indexed by a key. In particular, the adversary did not know which function from the family would be selected ahead of time. Because the number of functions in the family was exponentially large, one could not write down a polynomial-time adversary like the one above for a family.

However, the use of unkeyed functions which are collision resistant is going to be really important, so we need to be able to argue about them despite this definitional problem. So let us go back to our intuition: What we mean when we say a function is collision resistant is that we do not think the trivial adversary above can *be found* by the adversary attacking our system. In other words whilst we know that the trivial adversary exists, we do not think it humanly possible to construct it. We thus appeal to a concept which Rogaway calls *human ignorance*. We cannot define an advantage statement for such functions but we can define a notion of security.

Definition 14.1. *A function H is said to be collision resistant (by human ignorance) or* HI-CR *secure if it is believed to be infeasible to write down a collision for the function, i.e. two elements in the domain mapping to the same element in the codomain.*

It is harder to construct collision resistant hash functions than one-way hash functions due to the birthday paradox. To find a collision of a hash function H, we can keep computing

$$H(m_1), H(m_2), H(m_3), \ldots$$

until we get a collision. If the function has an output size of t bits then the probability of obtaining a collision after q queries to H is $q^2/2^{t+1}$. So we expact to obtain a collision after about $\sqrt{2^{t+1}}$ queries. This should be compared with the number of steps needed to find a preimage, which should be about 2^t for a well-designed hash function. Hence to achieve a security level of 128 bits for a collision resistant hash function we need roughly 256 bits of output.

Whilst the above, somewhat disappointing, definition of collision resistance is useful in many situations, in other situations a more robust, and less disappointing, definition is available. Here we consider a family of functions $\{f_k\}_{\mathbb{K}}$, the challenger picks a given function from the family and then asks the adversary to find a collision. We can define two security games, one where the adversary is given access to the function via an oracle (see Figure 14.3) and one where the adversary is actually given the key (see Figure 14.4).

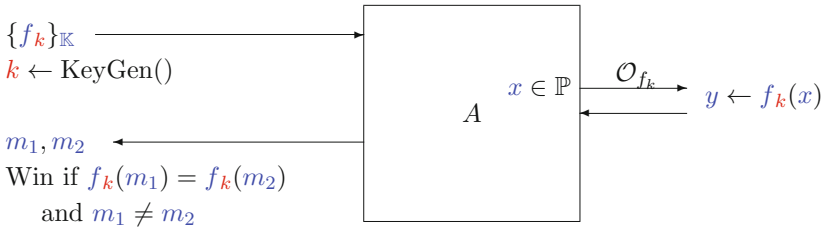


FIGURE 14.3. Security game for weak collision resistance of a family of functions

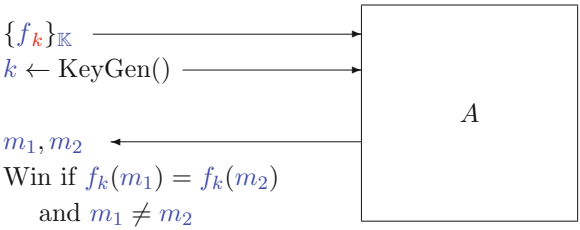


FIGURE 14.4. Security game for collision resistance of a family of functions

We define the advantages $\text{Adv}^{\mathsf{wCR}}_{\{f_k\}_{\mathbb{K}}}(A)$ and $\text{Adv}^{\mathsf{CR}}_{\{f_k\}_{\mathbb{K}}}(A)$ in the usual way as

$$\text{Adv}^{\mathsf{CR}}_{\{f_k\}_{\mathbb{K}}}(A) = \Pr[A \text{ wins the collision resistance game}],$$
$$\text{Adv}^{\mathsf{wCR}}_{\{f_k\}_{\mathbb{K}}}(A) = \Pr[A \text{ wins the weak collision resistance game}].$$

We say that the family is CR secure (resp. wCR secure) if the advantage is "small" for all adversaries A. For a *good* such function we hope that the advantage of the adversary in finding a collision is given by the birthday bound, i.e. $\frac{q^2}{2^{t+1}}$, where q models the number of queries A makes to the oracle $\mathcal{O}_{f_k}$, or essentially its running time in the case of collision resistance. Such keyed function families are often called (keyed) *collision resistant hash functions*.

In summary a cryptographic hash function needs to satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

(3) **Collision Resistant:** It should be hard to find two messages with the same hash value.

Note that the first two properties can also be applied to function families. We leave the respective definitions to the reader.

We can relate these three properties using reductions; note that the argument in the proof of the next lemma applies to collision resistance in any of the three ways we have defined it.

Lemma 14.2. *Assuming a function H is preimage resistant* for every element of the range of H *is a weaker assumption than assuming it is either collision resistant or second preimage resistant.*

PROOF. Suppose H is a function and let $\mathcal{O}$ denote an oracle which on input of y finds an x such that $H(x) = y$, i.e. $\mathcal{O}$ is an oracle which breaks the preimage resistance of the function H. Using $\mathcal{O}$ we can then find a collision in H by choosing x at random and then computing $y = H(x)$. Passing y to the oracle $\mathcal{O}$ will produce a value x' such that $y = H(x')$. Since H is assumed to have (essentially) infinite domain, it is unlikely that we have $x = x'$. Hence, we have found a collision in H. A similar argument applies to breaking the second preimage resistance of H. □

We can construct hash functions which are collision resistant but are not one-way for some of the range of H. As an example, let $g(x)$ denote a HI-CR secure function with outputs of bit length n. Now define a new hash function $H(x)$ with output size $n + 1$ bits as follows:

$$H(x) \leftarrow \begin{cases} 0\|x & \text{If } |x| = n, \\ 1\|g(x) & \text{Otherwise.} \end{cases}$$

The function $H(x)$ is clearly still HI-CR secure, as we have assumed $g(x)$ is HI-CR secure. But the function $H(x)$ is not preimage resistant as one can invert it on any value in the range which starts with a zero bit. So even though we can invert the function $H(x)$ on some of its input we are unable to find collisions.

Lemma 14.3. *Assuming a function is second preimage resistant is a weaker assumption than assuming it is collision resistant.*

PROOF. Assume we are given an oracle $\mathcal{O}$ which on input of x will find x' such that $x \neq x'$ and $H(x) = H(x')$. We can clearly use $\mathcal{O}$ to find a collision in H by choosing x at random. □

Another use of hash functions in practice will be for deriving keys from so-called keying material. When used in this way we say the function is a key derivation function, or KDF[1]. A key derivation function should act much like a PRF, except we now deal with arbitrary length inputs and *outputs*. Thus a KDF acts very much like a stream cipher with a fixed IV. We think of a keyed KDF G_k as taking a length ℓ, where ℓ defines how many bits of output are going to be produced for this key k. The key size for a KDF should also be variable, in that it can be drawn from any distribution of the challenger's choosing. Thus in our security game, in Figure 14.5, the challenger picks the distribution $\mathbb{K}$ and passes this to the KeyGen() function *and* the adversary. In the game the oracle $\mathcal{O}_{G_k}$ can only be called once.

In some sense we have already seen a KDF when we discussed CTR Mode; however for CTR Mode the key size was limited to the key size of the underlying block cipher, the output values of x in the game were limited to the size of the input block, *and* the output size had to be a multiple of a block length. So whilst CTR Mode *seems* to give us all the security properties we require, the functionality is rather limited. Despite this we will see later how to utilize CTR Mode to give us precisely the KDFs we require.

[1]We will see that we can constuct KDFs from other primitives, and not only hash functions.

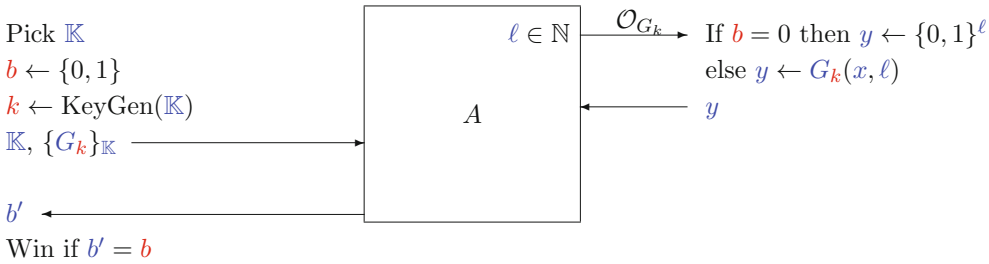


FIGURE 14.5. The security game for a KDF

We end this section with a final remark on how should we think about the security of (unkeyed) hash functions. The method adopted in much of practical cryptography is to assume that an unkeyed hash functions acts like a truly random function, and that (despite the adversary having the code of H) it therefore "acts" like a random oracle (see Section 11.9). In practice we define our cryptographic schemes and protocols assuming a random oracle, and then replace the random oracle in the real protocol by a fixed hash function. Whilst not a perfect methodology, this does work in most instances and results in relatively simple final schemes and protocols.

14.2. Padding

In Chapter 13 we skipped over discussing how to pad a message to a multiple of the block length; this is done via a padding scheme. In this chapter padding schemes will be more important, because we want to deal with arbitrary length messages, but the primitives from which we will build things from only take a fixed-length input, or an input which is a multiple of a fixed length. As when we discussed block ciphers we will call this fixed length the block size, and denote it by b. We will assume for simplicity that $b \geq 64$ in what follows.

Now given an input message m which is ℓ bits in length we will want to make it a message of $k \cdot b$ bits in length. This is done by padding, or extending, the message by adding extra bits onto the message until it is of the required length. However, there are many ways of doing this; we shall outline five. As a notation we write for the padded message

$$m \| \mathsf{pad}_i(|m|, b)$$

where i refers to the padding scheme (see below), and the padding function takes as input the length of the message and the block size we need to pad to. We always assume (in this book) that we want to pad to the next available block boundary given the message length and the padding scheme, and that the padding will be applied *at the end of the message*. This last point is not needed in theory, and indeed in theory one can obtain very efficient schemes by padding at the start. However, in practice almost all padding is applied to the end of a message.

We define our five padding schemes as follows:

- Method 0: Let v denote $b - |m| \pmod b$. Add v zeros to the end of the message $|m|$, i.e. $m \| \mathsf{pad}_0(|m|, b) = m \| 0^*$.
- Method 1: Let v denote $b - (|m| + 1) \pmod b$. Append a single 1 bit to the message, and then pad with v zeros, i.e. $m \| \mathsf{pad}_1(|m|, b) = m \| 10^*$.
- Method 2: Let v denote $b - (|m| + 65) \pmod b$. Encode $|m|$ as a 64-bit integer ℓ. Append a single 1 bit to the message, and then pad with v zeros, and then append the 64-bit integer ℓ, i.e. $m \| \mathsf{pad}_2(|m|, b) = m \| 10^* \| \ell$.
- Method 3: Let v denote $b - (|m| + 64) \pmod b$. Encode $|m|$ as a 64-bit integer ℓ. Pad with v zeros, and then append the 64-bit integer ℓ, i.e. $m \| \mathsf{pad}_3(|m|, b) = m \| 0^* \| \ell$.

- Method 4: Let v denote $b - (|m| + 2) \pmod b$. Append a single 1 bit to the message, and then pad with v zeros, and then add a one-bit, i.e. $m \| \mathsf{pad}_4(|m|, b) = m \| 10^*1$.

Notice that in all of these methods, bar method zero, given a padded message it is easy to work out the original message. For method zero we cannot tell the difference between a message consisting of one zero bit and two zero bits! We will see later why this causes a problem. In addition, for methods one to four, if two messages are of equal length then the two pads produced are equal. In addition, for padding methods two and three, if the messages are not of equal length then the two pads are distinct.

Before proceeding we pause to note that *any* of these padding schemes can be used with our earlier symmetric encryption schemes based on block ciphers, excluding padding method zero. However, when considering the functions in this chapter the precise padding scheme will have an impact on the underlying properties of the functions, in particular the security.

14.3. The Merkle–Damgård Construction

In this section we describe the basic Merkle–Damgård construction of a hash function taking inputs of arbitrary length from a hash function which takes inputs of a fixed length. The building block (a hash function taking inputs of a fixed length) is called a compression function, and the construction is very much like a mode of operation of a block cipher. The construction was analysed in two papers by Merkle and Damgård, although originally in the context of unkeyed compression functions.

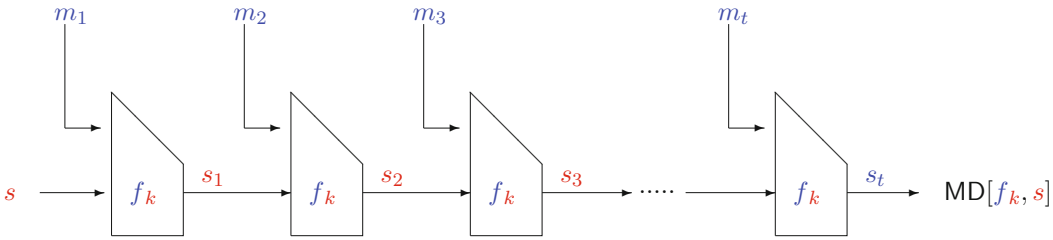


FIGURE 14.6. The Merkle–Damgård construction $\mathsf{MD}[f_k, s]$

See Figure 14.6 for an overview of the construction and Algorithm 14.1 for an algorithmic viewpoint. The construction is based on a family of compression functions f_k which map $(\ell + n)$-bit inputs to n-bit outputs. The construction also makes use of an internal state variable s_i which is updated by the application of the compression function. At each iteration this internal state is updated, by taking the current state and the next message block and applying the compression function. At the end the internal state is output as the result of the hash function, which we denote by $\mathsf{MD}[f_k, s]$.

Algorithm 14.1: Merkle–Damgård construction

Pad the input message m using a padding scheme, so that the output is a multiple of ℓ bits in length.
Divide the input m into t blocks of length ℓ bits. $m_1, \ldots, m_t$.
$s_0 \leftarrow s$.
for $i = 1$ **to** t **do**
  $s_i \leftarrow f_k(m_i \| s_{i-1})$.
return s_t.

14.3.1. Theoretical Properties of the Merkle–Damgård Construction: The problem with the $\mathsf{MD}[f_k, s]$ construction is that collision resistance depends on the padding scheme being used. The following toy example illustrates the problem. Suppose we have a function f_k which has $n = \ell = 4$ in the above notation, i.e. it takes as input bit strings of length eight bits, and outputs bit strings of length four.

Consider applying the following function to the messages, one which is one bit long and one which is two bits long.

$$m = 0b0, \ m' = 0b00.$$

The output of the basic Merkle–Damgård construction, when used with padding method zero, will be

$$f_k(m\|\mathsf{pad}_0(1,4)\|s_0) = f_k(0b0000) = f_k(m'\|\mathsf{pad}_0(2,4)\|s_0),$$

i.e. we obtain a collision. The problem is that padding method zero does not provide a unique way of decoding to obtain the original message. Thus $0b0000$ could correspond to the message $0b0$, or $0b00$, or $0b000$ or even $0b0000$. So when using the function $\mathsf{MD}[f_k, s]$ one should always use padding method one, two or three. In practice, all the standardized hash functions based on the Merkle–Damgård construction use padding method two. The use of padding method two will be exploited in the proof of Theorem 14.4 below.

Theorem 14.4. *Let $H_{k,s}(m) = \mathsf{MD}[f_k, s](m)$ denote the keyed hash function constructed using the Merkle–Damgård method from the keyed compression function $\{f_k(x)\}_{\mathbb{K}}$ family as above, using padding method two. Then if $\{f_k(x)\}_{\mathbb{K}}$ is CR/wCR secure, then so is $H_{k,s}$.*

Proof. Suppose A is an adversary against the CR/wCR security of $H_{k,s}(m)$. From A we wish to build an adversary B against the CR/wCR security of the family $\{f_k(x)\}_{\mathbb{K}}$. Algorithm B will either have as input k (for the CR game) or have access to an oracle to compute f_k for a fixed value of k (for the wCR game). Algorithm B then picks a random s and passes it to A (note that if we are assuming s is fixed and public then this step can be missed). In addition B either provides A with k, or (in the case of wCR security) provides an oracle for $H_{k,s}(m)$ created from B's own access to the oracle for computing f_k.

The adversary A will output, with some non-zero probability, a collision $H_{k,s}(m) = H_{k,s}(m')$, for which $m \neq m'$. Let us assume that m is t blocks long and m' is t' blocks long, and that (without loss of generality) the final added padding block does not produce a new block. So the actual messages hashed are

$$m_1, m_2, \ldots, m_t\|\mathbf{pb} \text{ and } m'_1, m_2, \ldots, m'_{t'}\|\mathbf{pb}',$$

where $\mathbf{pb}$ and $\mathbf{pb}'$ denote the specific public padding blocks added at the end of the messages.

We now unpeel the function $\mathsf{MD}[f_k, s]$ one layer at a time. We know that we have, since we have a collision, that

$$f_k(m_t\|\mathbf{pb}\|s_{t-1}) = f_k(m'_{t'}\|\mathbf{pb}'\|s_{t'-1}).$$

Now, unless $(m_t\|\mathbf{pb}\|s_{t-1}) = (m'_{t'}\|\mathbf{pb}'\|s'_{t-1})$ then we have that we have found a collision in f_k and algorithm B just outputs this collision. So suppose these tuples are equal, in particular that the last 64 bits of each of the padding blocks are equal. This last fact implies, since we are using padding method two, that the two messages are of equal length and so $t = t'$. It also means that the two chaining variables from the last round are equal and so we have a new equation to analyse

$$f_k(m_{t-1}\|s_{t-2}) = f_k(m'_{t-1}\|s'_{t-2}).$$

So we either have a collision now, or the two pairs of input are equal. Continuing in this way we either produce a collision on f_k *or* the two input messages are identical, i.e. $m = m'$, but we assumed this did not happen. Thus we must find a collision in f_k. □

The main problem, from a theoretical perspective, with the Merkle–Damgård construction is that we can think of it in one of three ways.

(1) In practice the value s is fixed to a given value IV, and the function f_k is not taken from a keyed function family, but is taken as a fixed function f. One then interprets Theorem 14.4 as saying that if the function f is HI-CR secure then $H(m) = \mathsf{MD}[f, IV]$ is also HI-CR secure.
(2) In practice we can also think of s_0 as defining a "key", but with the function f_k still being fixed. This viewpoint will be useful when defining HMAC below. In this case one interprets Theorem 14.4 as saying that if the function f is HI-CR secure then $H_s(m) = \mathsf{MD}[f, s]$ is wCR secure and CR secure.
(3) If we are able to select f_k from a family of pseudo-random functions, then we take s to be a fixed IV and Theorem 14.4 says that if the function f_k is CR secure then so is $H_k(m) = \mathsf{MD}[f_k, IV]$. Whilst this is the traditional result in theoretical cryptography, we note that it means absolutely nothing in practice.

Thus we have three ways of thinking of the Merkle–Damgård construction; two are useful in practice and one is useful in theory. So in this case theory and practice are not aligned.

Another property we will require of the compression function used within the Merkle–Damgård construction is that the fixed function $f(m\|s)$ is a secure message authentication code on $(\ell - 65)$-bit messages, when we consider s as the key to the MAC and padding method two is applied. This property will be needed when we construct HMAC below. We cannot prove this property for any of the specific instances of the function f considered below; it is simply an assumption, much like the assumption that AES defines a secure pseudo-random permutation.

We now turn to discussing the preimage and second preimage resistance of the Merkle–Damgård construction. To do this we make the following assumption about the function $f(m\|s)$, when considered as a function of two inputs m and s. This is non-standard and is made to make the following discussion slightly simpler.

Definition 14.5. *A function of two inputs $f(x, y)$ where $x \in X, y \in Y$ and $f(x, y) \in Z$ is said to be* uniformly distributed in its first component *if, for all values of y, the values of the function $f_y(x) = f(x, y)$ are uniformly distributed in Z as x ranges uniformly over X.*

This definition is somewhat reasonable to assume if the set X is much larger than Z, which it will be in all of the hash functions resulting from the Merkle–Damgård construction, and f is well designed. Using this we can show:

Theorem 14.6. *Let A be an adversary which finds preimages/second preimages for the hash function $H(m) = \mathsf{MD}[f, IV]$, assume that f is uniformly distributed in its first component, and that the first domain component is much larger than the codomain, then there is an adversary B which can find preimages/second preimages in f.*

Proof. We show the result for preimage resistance; for second preimage resistance we follow roughly the same argument. Let h be the input to the algorithm B. We pass h to the adversary A to obtain a preimage m of the hash function. Note that we can do this since f is uniformly distributed in its first component, and hence the value h "looks like" a value which could be output by the hash function. Hence, algorithm A will produce a preimage with its normal probability.

We now run the hash function forwards to obtain the input to the function f for the last round. This input is then output by algorithm B as its preimage on f. □

14.4. The MD-4 Family

The most widely deployed hash functions are MD-5, RIPEMD-160, SHA-1 and SHA-2, all of which are based on the Merkle–Damgård construction using a *fixed* (i.e. unkeyed) compression function f. The MD-5 algorithm produces outputs of 128 bits in size, whilst RIPEMD-160 and SHA-1

both produce outputs of 160 bits in length, whilst SHA-2 is actually three algorithms, SHA-256, SHA-384 and SHA-512, having outputs of 256, 384 and 512 bits respectively. All of these hash functions are derived from an earlier simpler algorithm called MD-4.

The seven main algorithms in the MD-4 family are

- **MD-4**: The function f has 3 rounds of 16 steps and an output bit length of 128 bits.
- **MD-5**: The function f has 4 rounds of 16 steps and an output bit length of 128 bits.
- **SHA-1**: The function f has 4 rounds of 20 steps and an output bit length of 160 bits.
- **RIPEMD-160**: The function f has 5 rounds of 16 steps and an output bit length of 160 bits.
- **SHA-256**: The function f has 64 rounds of single steps and an output bit length of 256 bits.
- **SHA-384**: The function f is identical to SHA-512 except the output is truncated to 384 bits, and the initial chaining value H is different.
- **SHA-512**: The function f has 80 rounds of single steps and an output bit length of 512 bits.

We discuss MD-4, SHA-1 and SHA-256 in detail; the others are just more complicated versions of MD-4, which we leave to the interested reader to look up in the literature. In recent years a number of weaknesses have been found in almost all of the early hash functions in the MD-4 family, for example MD-4, MD-5 and SHA-1. Hence, it is wise to move all application to use the SHA-2 algorithms, or the new sponge-based SHA-3 algorithm discussed later.

14.4.1. MD-4: We stress that MD-4 should be considered *broken*; we only present it for illustrative purposes as it is the simplest of all the constructions. In MD-4 there are three bit-wise functions on three 32-bit variables

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee ((\neg u) \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w. \end{aligned}$$

Throughout the algorithm we maintain a current hash state, corresponding to the value s_i in our discussion above.

$$H = (H_1, H_2, H_3, H_4)$$

of four 32-bit values. Thus the output length, is 128 bits long, with the input length to the compression function f being $512 + 128 = 640$ bits in length. There are various fixed constants (y_i, z_i, w_i), which depend on each round. We have

$$y_j = \begin{cases} 0 & 0 \leq j \leq 15, \\ \texttt{0x5A827999} & 16 \leq j \leq 31, \\ \texttt{0x6ED9EBA1} & 32 \leq j \leq 47. \end{cases}$$

and the values of z_i and w_i are given by the following arrays,

$$\begin{aligned} z_{0\ldots15} &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], \\ z_{16\ldots31} &= [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15], \\ z_{32\ldots47} &= [0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15], \\ w_{0\ldots15} &= [3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19], \\ w_{16\ldots31} &= [3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13], \\ w_{32\ldots47} &= [3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15]. \end{aligned}$$

We then execute the steps in Algorithm 14.2 for each 16 words entered from the data stream, where a word is 32 bits long and $\lll$ denotes a bit-wise rotation to the left. The data stream is

first padded with padding method two, so that it is an exact multiple of 512 bits long. On each iteration of Algorithm 14.2, the data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$.

Algorithm 14.2: The MD-4 $f(X \| s_{r-1})$ compression function

$(A, B, C, D) \leftarrow s_{r-1} = (H_1, H_2, H_3, H_4)$.
/* Round 1 */
for $j = 0$ **to** 15 **do**
  $t \leftarrow A + f(B, C, D) + X_{z_j} + y_j$.
  $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
/* Round 2 */
for $j = 16$ **to** 31 **do**
  $t \leftarrow A + g(B, C, D) + X_{z_j} + y_j$.
  $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
/* Round 3 */
for $j = 32$ **to** 47 **do**
  $t \leftarrow A + h(B, C, D) + X_{z_j} + y_j$.
  $(A, B, C, D) \leftarrow (D, t \lll w_j, B, C)$.
$(H_1, H_2, H_3, H_4) \leftarrow s_r = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$.

After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4.$$

When used in practice the initial value s_0 is initialized with the fixed values

$$H_1 \leftarrow \texttt{0x67452301}, \qquad H_2 \leftarrow \texttt{0xEFCDAB89},$$
$$H_3 \leftarrow \texttt{0x98BADCFE}, \qquad H_4 \leftarrow \texttt{0x10325476}.$$

14.4.2. SHA-1: When discussing SHA-1 it becomes clear that it is very, very similar to MD-4, for example we use the same bit-wise functions f, g and h as in MD-4. However, for SHA-1 the internal state of the algorithm is a set of five, rather than four, 32-bit values

$$H = (H_1, H_2, H_3, H_4, H_5),$$

resulting in a key/output size of 160 bits; the input data size is still kept at 512 bits though. We now only define four round constants y_1, y_2, y_3, y_4 via

$$y_1 = \texttt{0x5A827999},$$
$$y_2 = \texttt{0x6ED9EBA1},$$
$$y_3 = \texttt{0x8F1BBCDC},$$
$$y_4 = \texttt{0xCA62C1D6}.$$

After padding method two is applied, the data stream is loaded 16 words at a time into X_j for $0 \leq j < 16$, although note that internally the algorithm uses an expanded version of X_j with indices from 0 to 79. We then execute the steps in Algorithm 14.3 for each 16 words entered from the data stream. After all data has been read in, the output is the concatenation of the final value of

$$H_1, H_2, H_3, H_4, H_5.$$

Note the one-bit left rotation in the expansion step; an earlier algorithm called SHA (now called SHA-0) was initially proposed by NIST which did not include this one-bit rotation. However, this was soon replaced by the new algorithm SHA-1. It turns out that this single one-bit rotation

Algorithm 14.3: The SHA-1 $f(X\|s_{r-1})$ compression function

$(A, B, C, D, E) \leftarrow s_{r-1} = (H_1, H_2, H_3, H_4, H_5)$.
/* Expansion */
for $j = 16$ **to** 79 **do**
  $X_j \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \lll 1)$.
/* Round 1 */
for $j = 0$ **to** 19 **do**
  $t \leftarrow (A \lll 5) + f(B, C, D) + E + X_j + y_1$.
  $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 2 */
for $j = 20$ **to** 39 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$.
  $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 3 */
for $j = 40$ **to** 59 **do**
  $t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$.
  $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
/* Round 4 */
for $j = 60$ **to** 79 **do**
  $t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$.
  $(A, B, C, D, E) \leftarrow (t, A, B \lll 30, C, D)$.
$(H_1, H_2, H_3, H_4, H_5) \leftarrow s_r = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$.

improves the security of the resulting hash function quite a lot, since SHA-0 is now considered broken, whereas SHA-1 is considered just about alright (but still needing to be replaced).

To obtain the standardized version of SHA-1, the initial state s_0 is initialized with the values

$$H_1 \leftarrow \texttt{0x67452301}, \quad H_2 \leftarrow \texttt{0xEFCDAB89},$$
$$H_3 \leftarrow \texttt{0x98BADCFE}, \quad H_4 \leftarrow \texttt{0x10325476},$$
$$H_5 \leftarrow \texttt{0xC3D2E1F0}.$$

14.4.3. SHA-2: We only present the details of the SHA-256 variant; the others in the SHA-2 family are relatively similar. Unlike the other members of the MD-4 family, the SHA-2 algorithms consist of a larger number of rounds, each of one step. For an arbitrary input message m, SHA-256 produces a 256-bit message digest (or hash). The length l of the message m is bounded by $0 \leq l < 2^{64}$, due to the use of the standard MD-4 family padding procedure, namely what we have called padding method two.

SHA-256 processes the input message block by block, where each application of the $f_k(m)$ function is a function of 64 iterations of a single step. The step function makes use of slightly different f and g functions than those used in MD-4 and SHA-1. The SHA-2 f and g functions are given by

$$f'(u, v, w) = (u \wedge v) \oplus ((\neg u) \wedge w),$$
$$g'(u, v, w) = (u \wedge v) \oplus (u \wedge w) \oplus (v \wedge w).$$

SHA-2 also makes use of the following functions on single 32-bit words

$$\begin{aligned}\sum_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22),\\ \sum_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25),\\ \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3),\\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10),\end{aligned}$$

where $\ggg$ denotes right rotate, and $\gg$ denotes shift right. There are 64 constant words $K_0, \ldots, K_{63}$, which represent the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

For SHA-256 the internal state of the algorithm is a set of eight 32-bit values

$$H = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8),$$

corresponding to 256 bits of key/output. Again the input is processed 512 bits at a time. The data stream is loaded 16 words at a time into X_j for $0 \le j < 16$, which are then expanded to 64 words as in Algorithm 14.4. After all data have been read in, the output is the concatenation of the final values of

$$H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8.$$

The standard defines the initial state s_0 for SHA-256 to be given by setting the initial values to

Algorithm 14.4: The SHA-256 $f(X\|s_{r-1})$ compression function

$(A, B, C, D, E, F, G, H) \leftarrow (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8)$.
/* Expansion */
for $j = 17$ **to** 64 **do**
  $X_i \leftarrow \sigma_1(X_{i-2}) + X_{i-7} + \sigma_0(X_{i-15}) + X_{i-16}$.
/* Round */
for $j = 1$ **to** 64 **do**
  $t_1 \leftarrow H + \sum_1(E) + f'(E, F, G) + K_i + X_i$.
  $t_2 \leftarrow \sum_0(A) + g'(A, B, C)$.
  $(A, B, C, D, E, F, G, H) \leftarrow (t_1 + t_2, A, B, C, D + t_1, E, F, G)$.
$(H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8) \leftarrow s_r =$
$(H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E, H_6 + F, H_7 + G, H_8 + H)$.

$$\begin{aligned} H_1 &\leftarrow \texttt{0x6A09E667}, & H_2 &\leftarrow \texttt{0xBB67AE85},\\ H_3 &\leftarrow \texttt{0x3C6EF372}, & H_4 &\leftarrow \texttt{0xA54FF53A},\\ H_5 &\leftarrow \texttt{0x510E527F}, & H_6 &\leftarrow \texttt{0x9B05688C},\\ H_7 &\leftarrow \texttt{0x1F83D9AB}, & H_8 &\leftarrow \texttt{0x5BE0CD19}. \end{aligned}$$

14.5. HMAC

It is very tempting to define a MAC function from an unkeyed hash function as

$$t = H(k\|m\|\mathsf{pad}_i(|k| + |m|, b))$$

where k is the key for the MAC. After all a hash function should behave like a random oracle, and a random oracle by definition will be a secure MAC[2]. However, if we use the Merkle–Damgård construction for H then there is a simple attack. Let $\mathsf{MD}[f, s]$ denote one of the standardized Merkle–Damgård-based hash functions with a fixed compression function f and a fixed IV s. For

[2]It is perhaps worth proving this to yourself using the earlier games for a MAC and a random oracle.

simplicity assume the key k for the MAC is one block long, i.e. ℓ bits in length. The adversary first obtains a MAC t on the message m, by asking for

$$t = \mathsf{MD}[f, s]\left(k \middle\| m \middle\| \mathsf{pad}_i(\ell + |m|, \ell)\right).$$

We now know the state of the Merkle–Damgård function at this point, namely t. The adversary can then on his own compute the tag on any message of the form

$$m \middle\| \mathsf{pad}_i(\ell + |m|, \ell) \middle\| m'$$

for an m' of the adversaries choice. To get around this problem we define a function called HMAC, for Hash-MAC. Since HMAC is specifically designed to avoid the above problem with the Merkle–Damgård construction, it only makes sense to use it with hash functions created in this way. To define HMAC though we first create a related MAC which is easier to analyse called NMAC.

14.5.1. NMAC: *Nested MAC*, called NMAC, is built from two keyed hash functions F_{k_1} and G_{k_2}. The function NMAC is then defined by

$$\mathsf{NMAC}_{k_1,k_2}(m) = F_{k_1}\left(G_{k_2}(m)\right).$$

In particular we assume that $F_{k_1}(x)$ corresponds to a single application of a Merkle–Damgård (unkeyed) compression function f with input size ℓ, output size n, such that $k_1 \in \{0,1\}^\ell$ and $|x| + n + 66 \leq \ell$, and initial state k_1, but with a slightly strange padding method, namely

$$F_{k_1}(x) = f\left(\left(x \middle\| \mathsf{pad}_2(\ell + |x|, \ell)\right) \| k_1\right) = \mathsf{MD}[f, k_1]^*(x).$$

We let $\mathsf{MD}[f, k_1]^*$ denote the function $\mathsf{MD}[f, k_1](x)$ with this slightly modified padding formula. We assume that F_{k_1} is a secure message authentication code, which recall was one of our assumptions on the Merkle–Damgård compression functions we described above.

The function G_{k_2} is a wCR secure hash function, which produces outputs of size b with $b + n + 66 \leq \ell$. In practice we will take $b = n$, since we will construct G_{k_2} out of the same basic compression function f, with the same modified padding scheme above (namely the length gets encoded by adding an extra ℓ bits). Thus

$$G_{k_2}(m) = \mathsf{MD}[f, k_2]^*(m).$$

It is easily checked that with this modified padding method Theorem 14.4 still applies, and we can hence conclude that G_{k_2} is indeed a wCR secure hash function assuming f is HI-CR secure. Thus we have

$$\mathsf{NMAC}_{k_1,k_2}(m) = \mathsf{MD}[f, k_1]^*\left(\mathsf{MD}[f, k_2]^*(m)\right).$$

We then have the following theorem.

Theorem 14.7. *The message authentication code* NMAC *is* EUF-CMA *secure assuming* F_{k_1} *is a* EUF-CMA *secure MAC on* b*-bit messages and* G_{k_2} *is a* wCR *secure hash function outputting* b*-bit messages.*

Proof. Let A be the adversary against NMAC. We will use A to construct an adversary B against the MAC function F_{k_1}. Algorithm B has no input, but has access to an oracle which computes F_{k_1} for her. First B generates a random k_2, sets a list $\mathcal{L}$ to $\emptyset$ and then calls algorithm A.

When algorithm A queries its MAC oracle on input m, algorithm B responds by first computing the application of G_{k_2} on m to obtain y. This is then suitably padded and passed to the oracle provided to algorithm B to obtain t. Thus algorithm B obtains an NMAC tag on m under the key (k_1, k_2), and passes this back to algorithm A. Before doing so it appends the pair (y, t) to the list $\mathcal{L}$.

If A outputs a forgery (m^*, t^*) on NMAC then algorithm B computes $y^* = G_{k_2}(m^*)$. If there exists $(y^*, t^*) \in \mathcal{L}$ then B aborts, otherwise B returns the pair (y^*, t^*) as its EUF-CMA forgery for F_{k_1}.

Let ϵ_A denote the probability that A wins, and ϵ_B the probability that B wins. We have

$$\epsilon_A \leq \epsilon_B + \Pr[B \text{ aborts}].$$

We now note that the algorithm B could also be used to find collisions in G_{k_2} for a secret value k_2. Instead of picking k_2 at random we pick k_1, and then call A and respond using the oracle for G_{k_2} and then applying F_{k_1} for the known k_1. Thus $\Pr[B \text{ aborts}]$ is bounded by the probability of breaking the wCR security of G_{k_2}.

Hence, if F_{k_1} is an EUF-CMA secure MAC and G_{k_2} is wCR secure, then the two probabilities ϵ_B and $\Pr[B \text{ aborts}]$ are "small", and hence so is ϵ_A and so NMAC is secure. □

14.5.2. Building HMAC from NMAC: We can now build the function HMAC from our simpler function NMAC. The function HMAC is built from a standardized hash function given by $H(m) = \mathsf{MD}[f, IV]$ It makes use of two padding values opad (for outer pad) and ipad (for inner pad). These are ℓ-bit fixed values which are defined to be the byte 0x36 repeated $\ell/8$ times for opad and the byte 0x5C repeated $\ell/8$ times for ipad. The keys to HMAC are a single ℓ-bit value k.

We then define

$$\begin{aligned}\mathsf{HMAC}_k(m) &= H\left((k \oplus \mathsf{opad}) \,\|\, H\left((k \oplus \mathsf{ipad}) \,\|\, m\right)\right) \\ &= \mathsf{MD}[f, IV]\left((k \oplus \mathsf{opad}) \,\|\, \mathsf{MD}[f, IV]\left((k \oplus \mathsf{ipad}) \,\|\, m\right)\right).\end{aligned}$$

If we set $k_1 = f(k \oplus \mathsf{opad})$ and $k_2 = f(k \oplus \mathsf{ipad})$ then we have

$$\mathsf{HMAC}_{k_1,k_2}(m) = \mathsf{MD}[f, k_1]^*\left(\mathsf{MD}[f, k_2]^*(m)\right) = \mathsf{NMAC}_{k_1,k_2}(m).$$

Thus HMAC *is* a specific instance of NMAC, where the keys are derived in a very special manner. Hence we also require that the output of $f(m \| IV)$ acts like a weak form of pseudo-random function.

The astute reader will have noticed that when instantiated with SHA-256 the proof of HMAC will not apply, since the outer application of SHA-256 is performed on a message of three blocks, one for the key $k \oplus \mathsf{opad}$, one on the output of the inner application of SHA-256, and one on the padding block. The above proof of reduction to NMAC can clearly be modified to cope with this, with some additional assumptions and modifications to the NMAC proof. However, the added complications produce no extra insight, so we do not pursue them here. The interested reader should also note that there is a more elaborate proof of the HMAC construction which assumes even less of the components used to define the underlying Merkle–Damgård hash function.

14.6. Merkle–Damgård-Based Key Derivation Function

It is relatively straightforward to define KDFs given an (unkeyed) hash function from the Merkle–Damgård family. Recall that a KDF should take an arbitrary length input string and produce an arbitrary length output string which should look pseudo-random. There are two basic ways of doing this in the literature; we just give the basic ideas behind these constructions given a fixed hash function H of output length t bits and block size b. Let the number of output bits required from the KDF be n and set $\mathsf{cnt} = \lceil n/t \rceil$. We let $\mathsf{trunc}_n(m)$ denote the truncation of the message m to n bits in length, by removing bits to the left (the most significant bits), and let $\langle i \rangle_v$ denote the encoding of the integer i in v bits.

Method 1: This is a relatively basic method, whose security rests on assuming that H itself acts like a random oracle.

$$\mathsf{KDF}(m) = \mathsf{trunc}_n\Big(H\left(m\|\langle \mathsf{cnt}-1\rangle_{64}\right) \parallel H\left(m\|\langle \mathsf{cnt}-2\rangle_{64}\right) \parallel \cdots \parallel H\left(m\|\langle 1\rangle_{64}\right) \parallel H\left(m\|\langle 0\rangle_{64}\right)\Big).$$

Thus the ith output block is given by $H\left(m\|\langle i\rangle_{64}\right)$, except for the $(\mathsf{cnt}-1)$st block which is given by $\mathsf{trunc}_{n \pmod t}\left(H\left(m\|\langle \mathsf{cnt}-1\rangle_{64}\right)\right)$. This acts like CTR Mode in some sense, and can be very efficient if we first pad m out to a multiple of b and then compute $k = \mathsf{MD}[f, IV](m)$ with no padding method applied, and then compute

$$\mathsf{KDF}(m) = \mathsf{trunc}_n\Big(f\left(\langle \mathsf{cnt}-1\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \parallel f\left(\langle \mathsf{cnt}-2\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \parallel \cdots \parallel f\left(\langle 1\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right) \parallel f\left(\langle 0\rangle_{64}\|\mathsf{pad}_2(64+|m|,b)\|k\right)\Big).$$

Method 2: The second method utilizes the fact that HMAC itself acts like a pseudo-random function, and that the proof of HMAC establishes this in a stronger way than assuming the underlying hash function is a random oracle. Thus the second method is based on HMAC and the ith output block is given by

$$k_i = \mathsf{HMAC}\left(m\|k_{i-1}\|\langle i \pmod{256}\rangle_8\right),$$

where k_{-1} is defined to be the zero string of t bits.

14.7. MACs and KDFs Based on Block Ciphers

In this section we show how MACs and KDFs can also be derived from block ciphers, in addition to the compression functions we considered in the last section.

14.7.1. Message Authentication Codes from Block Ciphers: Some of the most widely used message authentication codes in practice are based on the CBC Mode of symmetric encryption, and are called "CBC-MAC". However, this is a misnomer, as for all bar a limited number of applications the following construction on its own does not form a secure message authentication code. However, we will return later to see how one can form secure MACs from the following construction.

Using a b-bit block cipher to give a b-bit keyed hash function is done as follows:

- The message m is padded to form a series of b-bit blocks; in principle *any* of the previous padding schemes can be applied.
- The blocks are encrypted using the block cipher in CBC Mode with the zero IV.
- Take the final block as the MAC.

Hence, if the b-bit data blocks, after padding, are

$$m_1, m_2, \ldots, m_q$$

then the MAC is computed by first setting $I_1 = m_1$ and $O_1 = e_k(I_1)$ and then performing the following for $i = 2, 3, \ldots, q$:

$$I_i = m_i \oplus O_{i-1},$$
$$O_i = e_k(I_i).$$

The final value $t = O_q$ is then output as the result of the computation. This is all summarized in Figure 14.7, and we denote this function by $\mathsf{CBC\text{-}MAC}_k(m)$.

We first look at an attack against CBC-MAC with padding method zero. Suppose we have a MAC value t on a message

$$m_1, m_2, \ldots, m_q,$$

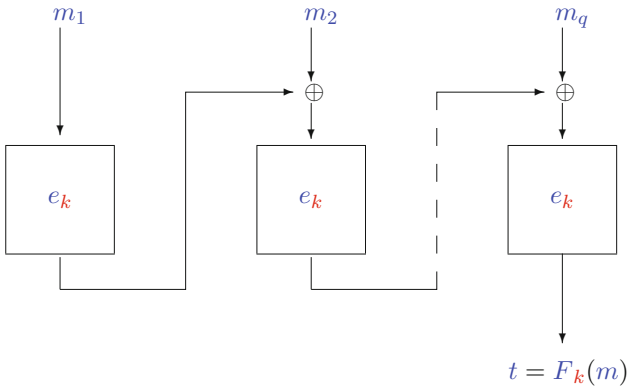


FIGURE 14.7. "CBC-MAC": flow diagram

consisting of a whole number of blocks. Then MAC tag t is also the MAC of the double length message

$$m_1, m_2, \ldots, m_q, t \oplus m_1, m_2, m_3, \ldots, m_q.$$

To see this notice that the input to the $(q+1)$st block cipher envocation is equal to the value of the MAC on the original message, namely t, exclusive-or'd with the $(q+1)$st block of the new message namely, namely $t \oplus m_1$. Thus the input to the $(q+1)$st cipher envocation is equal to m_1. This is the same as the input to the first cipher invocation, and so the MAC on the double length message is also equal to t.

One could suspect that use of more elaborate padding techniques would make attacks impossible; so let us consider padding method three. Let b denote the block length of the cipher and let $\mathbb{P}(n)$ denote the encoding within a block of the number n. To MAC a single block message m_1 one then computes

$$M_1 = e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right).$$

Suppose one obtains the MACs t_1 and t_2 on the single block messages m_1 and m_2. Then one requests the MAC on the three-block message

$$m_1, \mathbb{P}(b), m_3$$

for some new block m_3, obtaining the tag t_3, i.e.

$$t_3 = e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3 \cdot b)\right).$$

Now consider what is the MAC value on the three-block message

$$m_2, \mathbb{P}(b), m_3 \oplus t_1 \oplus t_2.$$

This tag is equal to t'_3, where

$$\begin{aligned}
t'_3 &= e_k\left(e_k\left(e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right) \oplus m_3 \oplus t_1 \oplus t_2\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= e_k\Big(e_k\Big(\underbrace{e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right)}\oplus m_3 \oplus \underbrace{e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right)} \oplus \underbrace{e_k\left(e_k(m_2) \oplus \mathbb{P}(b)\right)}\Big) \oplus \mathbb{P}(3 \cdot b)\Big) \\
&= e_k\left(e_k\left(m_3 \oplus e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right)\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= e_k\left(e_k\left(e_k\left(e_k(m_1) \oplus \mathbb{P}(b)\right) \oplus m_3\right) \oplus \mathbb{P}(3 \cdot b)\right) \\
&= t_3.
\end{aligned}$$

Hence, we see that on their own the non-trivial padding methods do not protect against MAC forgery attacks. However, if used in a one-time setting, i.e. for providing authentication to the

ciphertext in a data encapsulation mechanism; then the basic CBC-MAC construction is indeed secure.

It should be noted that if we put the length as the first block in the message then this padding method does produce a secure MAC. However, this is not used in practice since it requires the length of a message to be known before one has perhaps read it in. Thus in practice a different technique is used, the most popular of which is very similar to the NMAC construction above. We pick two block cipher keys k_1, k_2 at random and set

$$\mathsf{EMAC}_{k_1,k_2}(m) = e_{k_2}(\mathsf{CBC\text{-}MAC}_{k_1}(m)).$$

Just like NMAC and HMAC the inner function performs a MAC-like operation by sending a long message to a short-block-size message, and then the outer function uses the input to produce what looks like a random MAC value. However, we cannot reuse the proof of NMAC here, since that required the inner function to be weakly collision resistant which we already know CBC-MAC does not satisfy. Thus a new proof technique is needed.

Theorem 14.8. *EMAC is a secure message authentication code assuming the underlying block cipher e_k acts like a pseudo-random function. In particular let A denote an adversary against CBC-MAC which makes q queries of size at most ℓ blocks to its function oracle. Then there is an adversary B such that*

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{EMAC}}(A; q) \leq 2 \cdot \mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B) + \frac{2 \cdot T^2}{2^b} + \frac{1}{2^b},$$

where b is the block size of the block cipher e_k and $T = q \cdot \ell$.

Proof. The proof technique is very similar to the proof of Theorem 13.6, although far more intricate in analysis, thus we only sketch the details. Just like in the previous proof we switch from the actual block cipher family $\{e_k\}_{\mathbb{K}}$ to a truly random permutation, and then a truly random function. So from now on we assume that e_{k_1} is replaced by the random function f_1 and e_{k_2} is replaced by the random function f_2. As per the proof of Theorem 13.6, the adversary will only notice this has happened if she causes an output collision on one of the random functions. The factor of two in the theorem on the $\mathrm{Adv}^{\mathsf{PRP}}_{e_k}(B)$ term is due to the fact that we switch two block ciphers to random functions.

We note that a truly random function is a secure message authentication code by definition, and so all we need show now is that the output of EMAC behaves as a random function when the consistuent parts are random functions, irrespective of the strategy of the adversary. Thus we essentially need to show that the input to the outer layer function f_2 is itself random.

The only way the adversary could exploit the actual CBC-MAC definition to obtain a non-random output, which she could then exploit to create a forgery, would be to obtain a collision on the inputs to one of the calls to f_1 or f_2, and note she must do so without actually seeing the outputs to the calls, bar the final output of the EMAC function. This is where the intricate anaylsis comes in, which we defer to the paper references in the Further Reading section. □

14.7.2. Key Derivation Function from Block Ciphers: It is now relatively easy to define a key derivation function based on block ciphers. We use the fact that CBC-MAC or EMAC act like pseudo-random functions when applied to a long string; we then use the output to key a CTR Mode operation. For the "inner" compressing part of the key derivation function we can actually use CBC-MAC with the zero key. Thus we have that the ith block output by the KDF will be

$$e_k(\langle i \rangle_b)$$

where $k = \mathsf{CBC\text{-}MAC}_0(m)$.

14.8. The Sponge Construction and SHA-3

Having looked at how the Merkle–Damgård construction and block ciphers can be used to define various different cryptographic primitives, we now present a more modern technique to create hash functions, message authentication codes, key derivation functions and more. This modern technique is called the sponge construction. The two prior techniques use relatively strong components, namely PRPs and keyed compression functions, both of which satisfy relatively strong security requirements.

14.8.1. The Sponge Construction: The sponge construction takes a different approach; as its basic primitive it takes a *fixed* permutation p; such a permutation is clearly not one-way. Security is instead obtained by the method of chaining the permutation with the key, the input message and the padding scheme.

The entire construction is called a sponge, as the message is first entered into the sponge in a process akin to a sponge *absorbing* water. Then when we require output the sponge is *squeezed* to obtain as much output as we require. The sponge maintains an internal state of $r + c$ bits, and the permutation p acts as a permutation on the set $\{0,1\}^{r+c}$. The value r is called the *rate* of the sponge and the value c is called the *capacity*. The initial state is set to zero and then the message is entered block by block into the top r bits of the state, using exclusive-or. See Figure 14.8 for a graphical description of how a (keyed) sponge works. Note how padding method four is utilized; it is important that this is the padding method used in order to guarantee security if we use the same permutation in a sponge with different values for the rate. In particular there is no need to add length encodings as in the Merkle–Damgård construction. To define an unkeyed hash function one simply sets the key k to be the empty string.

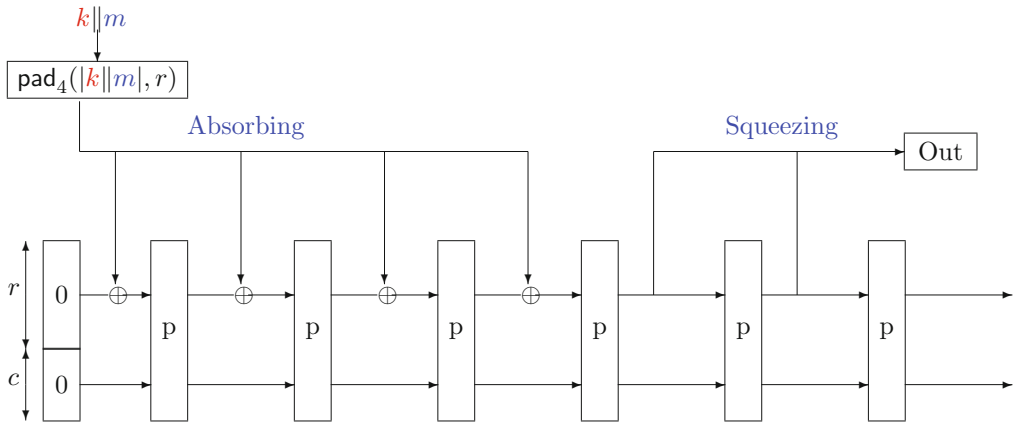


FIGURE 14.8. The sponge construction $\mathsf{Sp}[p]$

The idea is that as we squeeze the sponge we obtain r bits of output at a time. However, the output tells us nothing about the c hidden bits of the state. Thus to fully predict the next set of r output bits we appear to need to guess the c bits of missing state, or at least find a collision on these c bits of missing state. A careful analysis reveals that for a random permutation p the sponge construction $\mathsf{Sp}[p]$ has security equivalent to $2^{c/2}$ bits of symmetric cipher security.

One can show, using ideas and techniques way beyond what we can cover in this book, that the sponge construction, even for a zero key, cannot be *distinguished* from a random oracle (in some well-defined, but complicated sense). Recall from Chapter 11 that a random oracle is a function

which behaves like a random function, even though the adversary may have access to the function's "code".

14.8.2. SHA-3: SHA-3 was a competition organized, much like the AES competition, by NIST. The competition was launched in 2007 in response to the spectacular improvement in cryptanalyis of the MD-5 and SHA-1 algorithms in the preceding years. There were 64 competition entries, which were reduced to five by 2010, these being

- BLAKE, a proposal based on the ChaCha stream cipher.
- Grøstl, a Merkle–Damgård construction using components, such as the S-Box, from AES.
- JH, a sponge-like construction with a similar design philosophy to AES.
- Keccak, a sponge construction, and the eventual winner.
- Skein, a function based on the Threefish block cipher.

Keccak, designed by Joan Daemen, Guido Bertoni, Michaël Peeters and Gilles Van Assche, was declared the winner in October 2012.

The final SHA-3 function is a sponge-construction-based (unkeyed) hash function with a specific permutation p (which we shall now define), and with a zero-length key in the main sponge; see Figure 14.8[3]. The SHA-3 winner Keccak actually defines four different hash functions, with different output lengths (just as SHA-2 defines four different hash functions).

Being a sponge construction Keccak is parametrized by two values: the rate r and the capacity c. In Keccak the r and c values can be any values such that $r+c \in \{25, 50, 100, 200, 400, 800, 1600\}$, since we require $r+c$ to be equal to $25 \cdot 2^\ell$ for some integer value $\ell \in \{1, \ldots, 6\}$. This is due to the way the internal state of SHA-3 is designed, as we shall explain in a moment. If output hash sizes of 224, 256, 384 and 512 bits are required then the values of r should be chosen to be 1152, 1088, 832 and 576 repectively. If we go for the most efficient variant we want $r + c = 1600$, and then the associated capacities are 448, 512, 768 and 1024 respectively. If an arbitrary output length is required, for example for when used as a stream cipher or as a key derivation function, then one should use the values $(r, c) = (576, 1024)$.

The state of SHA-3 consists of a $5 \times 5 \times 2^\ell$ three-dimensional matrix of bits. We let $A[x, y, z]$ denote this array. If we fix y then the set $A[\cdot, y, \cdot]$ is called a plane, whereas if we fix z then the set $A[\cdot, \cdot, z]$ is called a slice, finally if we fix x then the set $A[x, \cdot, \cdot]$ is called a sheet. One-dimensional components in the x, y and z directions are called rows, columns and lanes respectively. See Figure 14.9.

The bit array $A[x, y, z]$ is mapped to a bit vector $a[i]$ using the convention $i = (5 \cdot y + x) \cdot 2^\ell + z$, for $x, y \in [0, 4]$ and $z \in [0, 2^\ell - 1]$, where, in the bit vector, bit 0 is in the leftmost position and bit $25 \cdot 2^\ell - 1$ is in the rightmost position. Thus the top r bits of the sponge construction state are bits $a[0]$ through to $a[r-1]$ and the bottom c bits are $a[r]$ through to $a[r+c-1]$.

All that remains to define SHA-3 is then to specify the permutation p. Just like the AES block cipher the construction is made up of repeated iteration of a number of very simple transformations. In particular there are five transformations called θ, ρ, π, χ and ι. The five functions are defined by, where if an index is less then zero or too large we assume a wrapping around:

- θ : For all x, y, z apply the transform, see Figure 14.10,

$$A[x, y, z] \leftarrow A[x, y, z] \oplus \sum_{y'=0}^{4} A[x-1, y', z] \oplus \sum_{y'=0}^{4} A[x+1, y', z-1].$$

[3]The author extends his thanks to Joan Daemen in helping with this section, and to the entire Keccak team for permission to include the figures found at http://keccak.noekeon.org/.

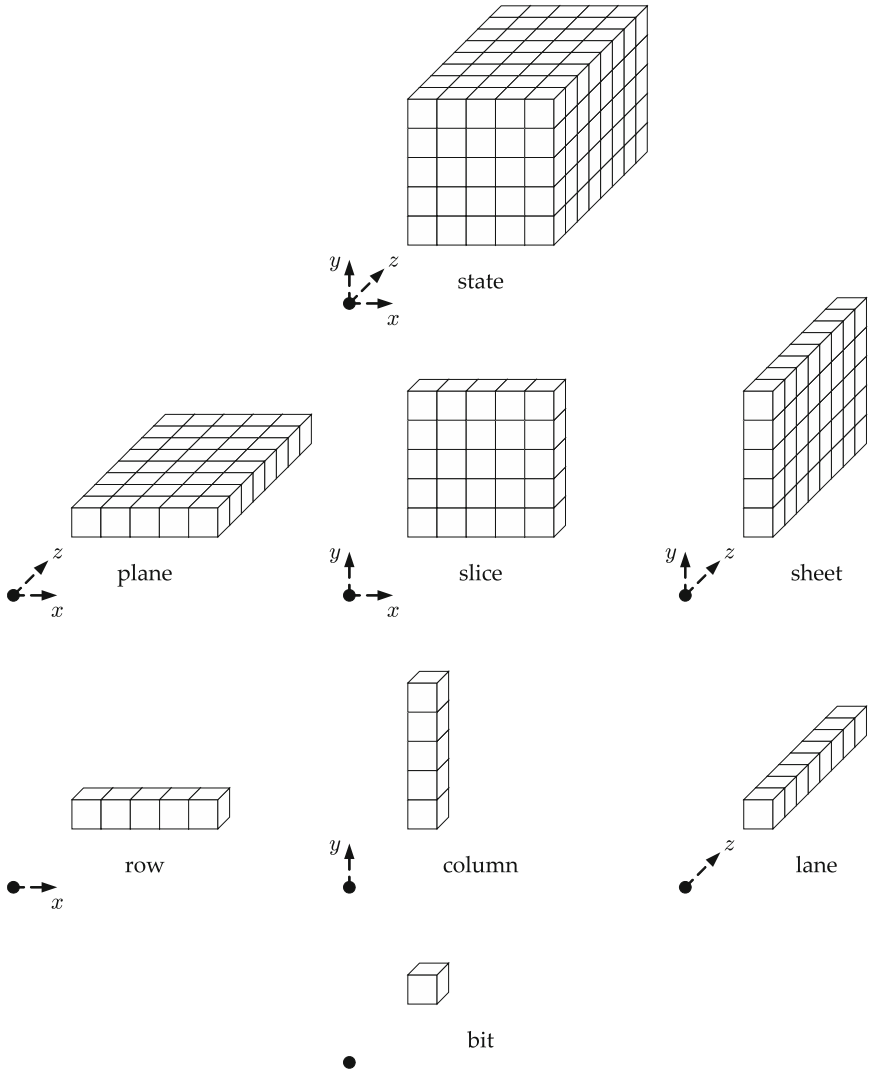


FIGURE 14.9. The SHA-3 state and its components

- ρ : For a given $(x, y) \in \mathbb{F}_5^2$, define $t \in \{0, \ldots, 23\}$ by the equation

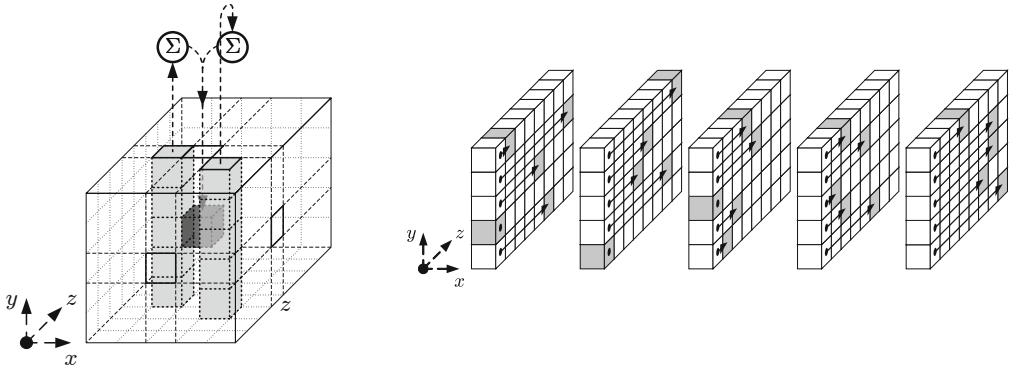
$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \pmod 5,$$

with $t = -1$ if $x = y = 0$. Then for all x, y, z apply the transform, see Figure 14.10,

$$A[x, y, z] \leftarrow A[x, y, (z - (t+1) \cdot (t+2)/2) \pmod{2^l}].$$

- π : For a given $(x, y) \in \mathbb{F}_5^2$ define (x', y') by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \pmod 5,$$



θ Transform ρ Transform

FIGURE 14.10. The SHA-3 θ and ρ transforms

then for all x, y, z apply the transform, see Figure 14.11,

$$A[x', y', z] \leftarrow A[x, y, z].$$

- χ : For all x, y, z apply the transform, see Figure 14.11,

$$A[x, y, z] \leftarrow A[x, y, z] \oplus (A[x + 1, y, z] + 1) \cdot A[x + 2, y, z].$$

  Notice that this is a non-linear operation, and is the only one we define for SHA-3.
- ι : For round number $i \in \{0, \ldots, 12 + 2 \cdot \ell - 1\}$ we define a round constant $\mathbf{rc}_i$. In round i the round constant $\mathbf{rc}_i$ is added to the $(0, 0)$-lane. The round constants, assuming the standard of 24 rounds, are

i	$\mathbf{rc}_i$	i	$\mathbf{rc}_i$	i	$\mathbf{rc}_i$	i	$\mathbf{rc}_i$
0	0x0000000000000001	1	0x0000000000008082	2	0x800000000000808A	3	0x8000000080008000
4	0x000000000000808B	5	0x0000000080000001	6	0x8000000080008081	7	0x8000000000008009
8	0x000000000000008A	9	0x0000000000000088	10	0x0000000080008009	11	0x000000008000000A
12	0x000000008000808B	13	0x800000000000008B	14	0x8000000000008089	15	0x8000000000008003
16	0x8000000000008002	17	0x8000000000000080	18	0x000000000000800A	19	0x800000008000000A
20	0x8000000080008081	21	0x8000000000008080	22	0x0000000080000001	23	0x8000000080008008

  These constants are truncated in the case that the lane is not 64 bits long.

The combination of these five transforms forms a *round*. Each round is repeated a total of $12+2\cdot\ell$ times to define the permutation p; for the standard configurations, with $\ell = 6$, this means the number of rounds is equal to 24. In Algorithm 14.5 we present an overview of the p function in SHA-3, which is called Keccak-f.

The key design principle is that we want to create an avalanche effect, meaning a small change in the state between two invocations should result in a massive change in the resulting output. Each of the five basic functions makes a small *local* change to the state, but combining different axes of the state. Thus, for example, the ρ transformation works by diffusion between the slices, much like ShiftRows works in AES. The π transform on the other hand works on each slice in turn, in a method reminiscent of MixColumns from AES, ι adds in "round constants" in a method reminiscent of AddRoundKey from AES, and χ provides non-linearity per round in much the same way as SubBytes does in AES. Finally, θ provides a further form of mixing between neighbouring columns. As all these steps are applied more than $2 \cdot \ell$ times every entry in the state affects every other entry in a non-linear manner.

14.8.3. Sponges With Everything: We now discuss a number of additional functionalities which arise from the basic sponge construction, thus showing its utility in designing other primitives. As one can see from all of these constructions, bar that of an IND-CCA secure encryption scheme,

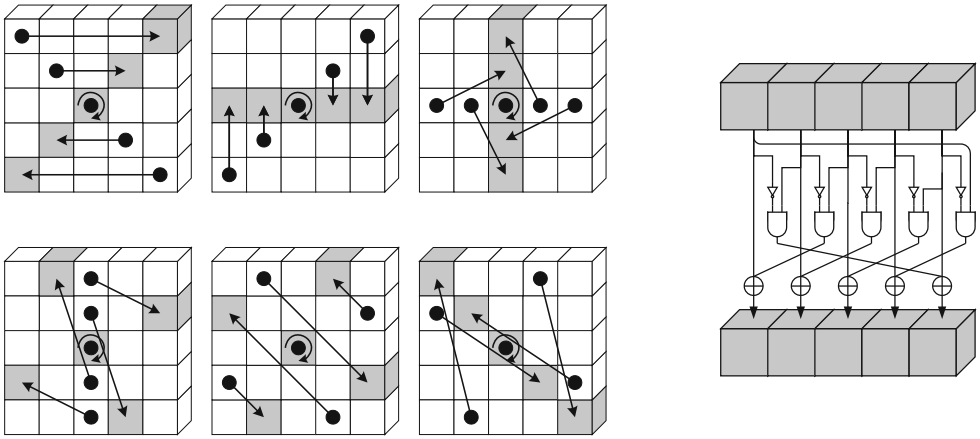


FIGURE 14.11. The SHA-3 π and χ transforms

Algorithm 14.5: The SHA-3 p function: Keccak-f

```
/* Permutation */
for i = 0 to 12 + 2 · ℓ − 1 do
    A ← θ(A).
    A ← ρ(A).
    A ← π(A).
    A ← χ(A).
    A ← ι(A, i).
```

they are less complicated than their equivalent block-cipher-based or compression-function-based cousins.

Sponge-Based MAC: Recall that message authentication codes are symmetric keyed primitives which ensure that a message is authentic, i.e. has not been tampered with and has come from some other party possessing the same symmetric key. If we take the key $k \in \{0,1\}^r$ and the message $m \in \{0,1\}^*$, and then apply our sponge construction, we obtain a MAC of any length we want, r bits at a time. Since the sponge "acts like" a random oracle the resulting code acts like a random value, and so the only way for an adversary to win the EUF-CMA game is to obtain a collision in the sponge construction when used in this way. Thus the sponge construction on its own, with a suitable choice of the permutation p, is a secure message authentication code.

Sponge-Based KDF/Stream Cipher: A sponge-based hash function can be utilized as a stream cipher, and hence a KDF, in a relatively simple way. We take the key k and append an IV if needed. The result is padded via padding method four, and then this is absorbed into the sponge. The keystream is then squeezed out of the sponge r bits at a time. Again, the fact that the sponge "acts like" a random oracle ensures that the keystream is truly random and with a random IV gives us an IND-CPA secure stream cipher.

Sponge-Based IND-CCA Secure Encryption: An interesting aspect of the sponge construction for hash functions is that the same construction can be used to produce an IND-CCA secure symmetric encryption scheme. This construction is a little more involved; it works by combining the above method for obtaining a stream cipher from a sponge, and the method for obtaining a MAC

from a sponge, in an Encrypt-then-MAC methodology. However, for efficiency, the tapping of the keystream and the feeding in of the ciphertext to obtain a MAC on the ciphertext happen simultaneously. To do this, and obtain security, a special form of padding must be used on the message; details are in the Further Reading section of this chapter. We present the (simplified) encryption and decryption operations in Figure 14.12.

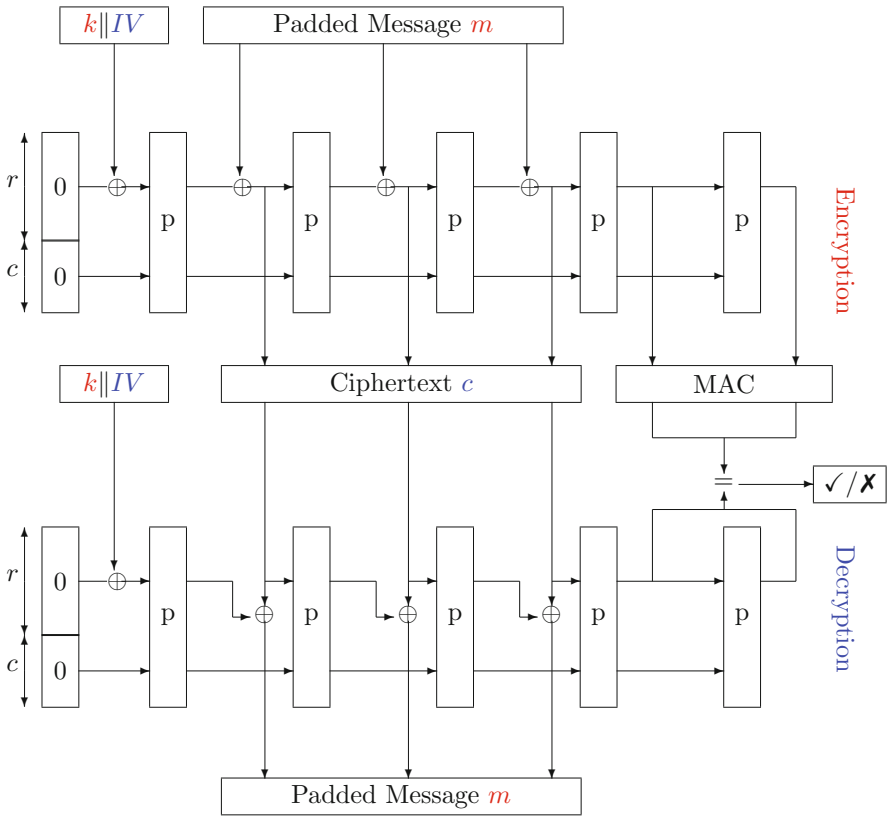


FIGURE 14.12. IND-CCA encryption (above) and decryption (below) using a sponge Sp[p]

Chapter Summary

- Keyed hash functions have a well-defined security model; a similar definition for unkeyed hash functions is harder to give. So we have to rely on "human ignorance".
- Due to the birthday paradox when collision resistance is a requirement, the output of the hash function should be at least twice the size of what one believes to be the limit of the computational ability of the attacker.
- Most hash functions are iterative in nature, although most of the currently deployed ones from the MD4 family have been shown to be weaker than expected.

- The current best hash functions are SHA-2, based on the Merkle–Damgård construction, and SHA-3, which is a so-called sponge construction.
- A message authentication code is in some sense a keyed hash function, whilst a key derivation function is a hash function with arbitrary length codomain.
- Message authentication codes and key derivation functions can be created out of either block ciphers or hash functions.

Further Reading

A detailed description of both SHA-1 and the SHA-2 algorithms can be found in the FIPS standard below; this includes a set of test vectors as well. The proof of security of EMAC is given in the paper by Petrank and Rackoff, where it is called DMAC. Details of SHA-3 can be found on the Keccak website given below.

FIPS PUB 180-4, *Secure Hash Standard (SHS)*. NIST, 2012.

G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. *The Keccak Sponge Function Family*. http://keccak.noekeon.org/.

E. Petrank and C. Rackoff. *CBC MAC for real-time data sources*. Journal of Cryptology, **13**, 315–338, 2000.

CHAPTER 15

The "Naive" RSA Algorithm

Chapter Goals

- To understand the naive RSA encryption algorithm and the assumptions on which its security relies.
- To do the same for the naive RSA signature algorithm.
- To show why these naive versions cannot be considered secure.
- To explain Wiener's attack on RSA using continued fractions.
- To explain how to use Coppersmith's Theorem for finding small roots of modular polynomial equations to extend this attack to other situations.
- To introduce the notions of partial key exposure and fault analysis.

15.1. "Naive" RSA Encryption

The RSA algorithm was the world's first public key encryption algorithm, and it has stood the test of time remarkably well. The RSA algorithm is based on the difficulty of the RSA problem considered in Chapter 2, and hence it is based on the difficulty of finding the prime factors of large integers. However, we have seen that it may be possible to solve the RSA problem without factoring, hence the RSA algorithm is not based completely on the difficulty of factoring.

Suppose Alice wishes to enable anyone to send her secret messages, which only she can decrypt. She first picks two large secret prime numbers p and q. Alice then computes

$$N = p \cdot q.$$

Alice also chooses an encryption exponent e which satisfies

$$\gcd(e, (p-1) \cdot (q-1)) = 1.$$

It is common to choose $e = 3, 17$ or $65\,537$. Now Alice's public key is the pair $\mathfrak{pk} = (N, e)$, which she can publish in a public directory. To compute her private key Alice applies the extended Euclidean algorithm to e and $(p-1)(q-1)$ to obtain the decryption exponent d, which should satisfy

$$e \cdot d = 1 \pmod{(p-1)(q-1)}.$$

Alice keeps secret her private key, which is the triple $\mathfrak{sk} = (d, p, q)$. Actually, she could simply throw away p and q, and retain a copy of her public key which contains the integer N, but as we saw in Chapter 6 holding onto the prime factors can aid the exponentiation algorithm modulo N.

Now suppose Bob wishes to encrypt a message to Alice. He first looks up Alice's public key and represents the message as a number m which is strictly less than the public modulus N. The ciphertext is then produced by raising the message to the power of the public encryption exponent modulo the public modulus, i.e.

$$c \leftarrow m^e \pmod{N}.$$

On receiving c Alice can decrypt the ciphertext to recover the message by exponentiating by the private decryption exponent, i.e.

$$m \leftarrow c^d \pmod{N}.$$

This works since the group $(\mathbb{Z}/N\mathbb{Z})^*$ has order

$$\phi(N) = (p-1)(q-1)$$

and so, by Lagrange's Theorem,

$$x^{(p-1)\cdot(q-1)} = 1 \pmod{N},$$

for all $x \in (\mathbb{Z}/N\mathbb{Z})^*$. Thus, for some integer s we have

$$ed - s \cdot (p-1) \cdot (q-1) = 1,$$

and so

$$\begin{aligned} c^d &= (m^e)^d = m^{e\cdot d} \\ &= m^{1+s(p-1)\cdot(q-1)} = m \cdot m^{s\cdot(p-1)\cdot(q-1)} \\ &= m. \end{aligned}$$

To make things clearer let's consider a baby example. Choose $p = 7$ and $q = 11$, and so $N = 77$ and $(p-1)\cdot(q-1) = 6 \cdot 10 = 60$. We pick as the public encryption exponent $e = 37$, since we have $\gcd(37, 60) = 1$. Then, applying the extended Euclidean algorithm we obtain $d = 13$ since

$$37 \cdot 13 = 481 = 1 \pmod{60}.$$

Suppose the message we wish to transmit is given by $m = 2$, then to encrypt m we compute

$$c \leftarrow m^e \pmod{N} = 2^{37} \pmod{77} = 51,$$

whilst to decrypt the ciphertext c we compute

$$m \leftarrow c^d \pmod{N} = 51^{13} \pmod{77} = 2.$$

The security of RSA on first inspection relies on the difficulty of finding the private encryption exponent d given only the public key, namely the public modulus N and the public encryption exponent e. In Chapter 2 we showed that the RSA problem is no harder than FACTOR, hence if we can factor N then we can find p and q and hence we can calculate d. Hence, if factoring is easy we can break RSA. Currently it is recommended that one takes public moduli of size around 2048 bits to ensure medium-term security.

Recall, from Chapter 11, that for a public key algorithm the adversary always has access to the encryption algorithm, hence she can always mount a chosen plaintext attack. We can show that RSA encryption meets our weakest notion of security for public key encryption, namely OW-CPA, assuming that the RSA problem is hard. To show this we use the reduction arguments of previous chapters. This example is rather trivial but we labour the point since these arguments are used over and over again.

Lemma 15.1. *If the RSA problem is hard then the naive RSA encryption scheme is* OW-CPA *secure. In particular if A is an adversary which breaks the* OW-CPA *security of naive RSA encryption for RSA moduli of v bits in length, then there is an adversary B such that*

$$\mathrm{Adv}^{\mathsf{OW\text{-}CPA}}_{\mathsf{RSA}(v)}(A) = \mathrm{Adv}_v\mathsf{RSA}(B).$$

Proof. We wish to give an algorithm which solves the RSA problem using an algorithm to break the RSA cryptosystem as an oracle. If we can show this then we can conclude that breaking the RSA cryptosystem is no harder than solving the RSA problem.

Recall that the RSA problem is given $N = p \cdot q$, e and $y \in (\mathbb{Z}/N\mathbb{Z})^*$, compute an x such that $x^e \pmod{N} = y$. We use our oracle to break the RSA encryption algorithm to "decrypt" the message

corresponding to $c = y$; this oracle will return the plaintext message m. Then our RSA problem is solved by setting $x \leftarrow m$ since, by definition,

$$m^e \pmod N = c = y.$$

So if we can break the RSA algorithm then we can solve the RSA problem. □

This is, however, the best we can hope for since naive RSA encryption is deterministic, and thus we have the following.

Lemma 15.2. *Naive RSA encryption is* not IND-CPA *secure.*

Proof. Recall from Chapter 11 that in the IND-CPA game the adversary produces two plaintexts, which we shall denote by m_0 and m_1. The challenger, then encrypts one of them to obtain the challenge ciphertext $c^* \leftarrow {m_b}^e \pmod N$, for some hidden bit $b \in \{0, 1\}$. The adversary's goal is then to determine b. This task can be easily accomplished since all the adversary needs to do is to compute $c \leftarrow m_1^e \pmod N$, and then

- if $c^* = c$ then the attacker knows that $m_b = m_1$,
- if $c^* \neq c$ then the attacker knows that $m_b = m_0$.

□

The problem is that the attacker has access to the encryption function; it is a public key scheme after all. But using a deterministic encryption function is not the only problem with RSA, since RSA is also malleable due to the homomorphic property.

Definition 15.3 (Homomorphic Property)**.** *An encryption scheme has the (multiplicative) homomorphic property if given the encryptions of m_1 and m_2 we can determine the encryption of $m_1 \cdot m_2$, without knowing m_1 or m_2.*

A similar definition can be given for additive homomorphisms as well. That RSA has the homomorphic property follows from the equation

$$(m_1 \cdot m_2)^e \pmod N = ((m_1{}^e \pmod N) \cdot (m_2{}^e \pmod N)) \pmod N.$$

One can use the homomorphic property to show that RSA is not even one-way secure under an adaptive chosen ciphertext attack.

Lemma 15.4. *Naive RSA encryption is* not OW-CCA *secure.*

Proof. Recall from Chapter 11 that the OW-CCA game is like the OW-CPA game, except that the adversary now has access to a decryption oracle which can decrypt any ciphertext, bar the target ciphertext c^*. Suppose the challenger gives the adversary the challenge ciphertext

$$c^* = (m^*)^e \pmod N.$$

The goal of the adversary is to find m^*. The adversary then creates the "related" ciphertext $c = 2^e \cdot c^*$ and asks her decryption oracle to decrypt c to produce m. Notice that this is a legal query under the rules of the game as $c \neq c^*$. The adversary can then compute

$$\begin{aligned} \frac{m}{2} &= \frac{c^d}{2} = \frac{(2^e \cdot c^*)^d}{2} \\ &= \frac{2^{e \cdot d} \cdot (c^*)^d}{2} = \frac{2 \cdot m^*}{2} = m^*. \end{aligned}$$

□

In Chapter 16 we will present variants of RSA encryption, as well as other public key encryption schemes, and show that they are secure in the sense of IND-CCA. It is these more advanced algorithms which one should use in practice, and this is why we have dubbed the above method of encrypting "naive" RSA.

15.1.1. Rabin Encryption: The obvious question is whether we can construct an OW-CPA secure scheme, which is efficient, and which is based on the factoring problem itself. The Rabin encryption scheme is one such scheme; it replaces the RSA map (which is a permutation on the group $(\mathbb{Z}/N\mathbb{Z})^*$) with a map which is not injective. The Rabin scheme, due to Michael Rabin, bases its security on the difficulty of extracting square roots modulo $N = p \cdot q$, for two large unknown primes p and q. Recall Theorem 2.4, which showed that this SQRROOT problem is polynomial-time equivalent to the factoring problem.

Despite these plus points the Rabin system is not used as much as the RSA system. It is, however, useful to study for a number of reasons, both historical and theoretical. The basic idea of the system is also used in some higher-level protocols.

Key Generation: We first choose prime numbers of the form $p = q = 3 \pmod 4$, since this makes extracting square roots modulo p and q very fast. The private key is then the pair $\mathfrak{sk} \leftarrow (p, q)$, and the public key is $\mathfrak{pk} \leftarrow N = p \cdot q$.

Encryption: To encrypt a message m using the above public key, in the Rabin encryption algorithm we compute $c \leftarrow m^2 \pmod N$. Hence, encryption involves one multiplication modulo N, and is therefore much faster than RSA encryption, even when one chooses a small RSA encryption exponent. Indeed, encryption in the Rabin encryption system is much faster than almost any other public key scheme.

Decryption: Decryption is far more complicated; essentially we want to compute the value of $m = \sqrt{c} \pmod N$. At first sight this uses no private information, but a moment's thought reveals that one needs the factorization of N to be able to find the square root. In particular to compute m one computes

$$m_p = \sqrt{c} \pmod p = c^{(p+1)/4} \pmod p = \pm 35,$$
$$m_q = \sqrt{c} \pmod q = c^{(q+1)/4} \pmod q = \pm 44,$$

and then combines m_p and m_q by the Chinese Remainder Theorem. There are however four possible square roots modulo N, since N is the product of two primes. Hence, on decryption we obtain *four* possible plaintexts. This means that we need to add redundancy to the plaintext before encryption in order to decide which of the four possible plaintexts corresponds to the intended one.

Example: Let the private key be given by $p = 127$ and $q = 131$, and the public key be given by $N = 16\,637$. To encrypt the message $m = 4\,410$ we compute

$$c = m^2 \pmod N = 16\,084.$$

To decrypt we evaluate the square root of c modulo p and q

$$m_p = \sqrt{c} \pmod p = \pm 35,$$
$$m_q = \sqrt{c} \pmod q = \pm 44.$$

Now we apply the Chinese Remainder Theorem to both $\pm 35 \pmod p$ and $\pm 44 \pmod q$ so as to find the square root of c modulo N,

$$s = \sqrt{c} \pmod N = \pm 4\,410 \text{ and } \pm 1\,616.$$

This leaves us with the four "messages"

$$1\,616,\ 4\,410,\ 12\,227,\ \text{ or } 15\,021.$$

It should be pretty clear that any adversary breaking the OW-CPA security of the Rabin encryption scheme can immediately be turned into an adversary to break the SQRROOT problem, and hence into an adversary to factor integers. So we have the following.

Theorem 15.5. *If A is an adversary against the* OW-CPA *security of the Rabin encryption scheme* Π *for moduli of size v bits, then there is an adversary B against the factoring problem with*

$$\mathrm{Adv}_{\Pi}^{\mathsf{OW\text{-}CPA}}(A) = 2 \cdot \mathrm{Adv}_{v}^{\mathsf{FACTOR}}(B),$$

the factor of two coming from Lemma 2.6.

It should also be pretty obvious that the scheme is not OW-CCA secure, since the scheme is obviously malleable in the same way that RSA was. In addition it is clearly not IND-CPA as it is deterministic, like RSA.

15.2. "Naive" RSA Signatures

The RSA encryption algorithm is particularly interesting since it can be used directly as a so-called signature algorithm with message recovery.

- The sender applies the RSA *decryption* transform to generate the signature, by taking the message and raising it to the private exponent d

$$s \leftarrow m^d \pmod{N}.$$

- The receiver then applies the RSA *encryption* transform to recover the original message

$$m \leftarrow s^e \pmod{N}.$$

But this raises the question; how do we check the validity of the signature? If the original message is in a natural language such as English then one can verify that the extracted message is also in the same natural language. But this is not a solution for all possible messages. Hence one needs to add redundancy to the message.

One, almost prehistoric, way of doing this in the early days of public key cryptography was the following. Suppose the message m is t bits long and the RSA modulus N is k bits long, with $t < k - 32$. We first pad m to the right with zeros to produce a string of length a multiple of eight. We then add $(k - t)/8$ bytes to the left of m to produce a byte-string

$$m \leftarrow 00\|01\|FF\|FF \ldots \|FF\|00\|m.$$

The signature is then computed via

$$m^d \pmod{N}.$$

When verifying the signature we ensure that the recovered value of m has the correct padding. This form of padding also seems to prevent the following trivial existential forgery attack. The attacker picks s at random and then computes

$$m \leftarrow s^e \pmod{N}.$$

The attacker then has the signature s on the message m.

Moreover, the padding scheme also seems to prevent selective forgeries, which are in some sense an even worse form of weakness. Without the padding scheme we can produce a selective forgery, using access to a signing oracle, as follows. Suppose the attacker wishes to produce a signature s on the message m. She first generates a random $m_1 \in (\mathbb{Z}/N\mathbb{Z})^*$ and computes

$$m_2 \leftarrow \frac{m}{m_1}.$$

Then the attacker asks her oracle to sign the messages m_1 and m_2. This results in two signatures s_1 and s_2 such that

$$s_i = {m_i}^d \pmod{N}.$$

The attacker can then compute the signature on the message m by computing

$$s \leftarrow s_1 \cdot s_2 \pmod{N}$$

since

$$\begin{aligned} s &= s_1 \cdot s_2 \pmod N \\ &= {m_1}^d \cdot {m_2}^d \pmod N \\ &= (m_1 \cdot m_2)^d \pmod N \\ &= m^d \pmod N. \end{aligned}$$

Here we have used once more the homomorphic property of the RSA function, just as we did when we showed that RSA encryption was not OW-CCA secure.

But not all messages will be so short so as to fit into the above method. Hence, naively to apply the RSA signature algorithm to a long message m we need to break it into blocks and sign each block in turn. This is very time-consuming for long messages. Worse than this, we must add serial numbers and more redundancy to each message otherwise an attacker could delete parts of the long message without us knowing, just as could happen when encrypting using a block cipher in ECB Mode. This problem arises because our signature model is one giving message recovery, i.e. the message is recovered from the signature and the verification process. If we used a model called a signature with *appendix* then we could first produce a hash of the message to be signed and then just sign the hash.

Using a cryptographic hash function H, such as those described in Chapter 14, it is possible to make RSA into a signature scheme without message recovery, which is very efficient for long messages. Suppose we are given a long message m for signing; we first compute $H(m)$ and then apply the RSA signing transform to $H(m)$, i.e. the signature is given by

$$s \leftarrow H(m)^d \pmod N.$$

The signature and message are then transmitted together as the pair (m, s). Verifying a message/signature pair (m, s) generated using a hash function involves three steps.

- "Encrypt" s using the RSA encryption function to recover h, i.e.

$$h \leftarrow s^e \pmod N.$$

- Compute $H(m)$ from m.
- Check whether $h = H(m)$. If they agree accept the signature as valid, otherwise the signature should be rejected.

Since a hash function does not usually have codomain the whole of the integers modulo N, in practice one needs to first hash and then pad the message. We could, for example, use the padding scheme given earlier when we discussed RSA with message recovery. If we assume the hash function produces a value in the range $[0, \ldots, N-1]$ then the above scheme is called RSA-FDH, for RSA *Full Domain Hash*. The name is because the codomain of the hash function is the entire domain of the RSA function.

Recall that when we discussed cryptographic hash functions we said that they should satisfy the following three properties:

(1) **Preimage Resistant:** It should be hard to find a message with a given hash value.
(2) **Collision Resistant:** It should be hard to find two messages with the same hash value.
(3) **Second Preimage Resistant:** Given one message it should be hard to find another message with the same hash value.

It turns out that all three properties are needed when using the RSA-FDH signing algorithm, as we shall now show.

Requirement for Preimage Resistance: The one-way property stops a cryptanalyst from cooking up a message with a given signature. For example, suppose we are using RSA-FDH but with a hash function which does not have the one-way property. We then have the following attack.

- The adversary computes
$$h \leftarrow r^e \pmod N$$
for some random integer r.
- The adversary also computes the preimage of h under H (recall we are assuming that H does not have the one-way property), i.e. Eve computes
$$m \leftarrow H^{-1}(h).$$

The adversary now has your signature (m, r) on the message m. Recall that such a forgery is called an existential forgery, since the attacker may not have any control over the contents of the message on which she has obtained a digital signature.

Requirement for Collision Resistance: This is needed to avoid the following attack, which is performed by the legitimate signer.

- The signer chooses two messages m and m' with $H(m) = H(m')$.
- They sign m and output the signature (m, s).
- Later they repudiate this signature, saying it was really a signature on the message m'.

As a concrete example one could have that m is an electronic cheque for 1 000 euros whilst m' is an electronic cheque for 10 euros.

Requirement for Second Preimage Resistance: This property is needed to stop the following attack.

- An attacker obtains your signature (m, s) on a message m.
- The attacker finds another message m' with $H(m') = H(m)$.
- The attacker now has your signature (m', s) on the message m'.

Thus, the security of any signature scheme which uses a cryptographic hash function will depend both on the security of the underlying hard mathematical problem, such as factoring or the discrete logarithm problem, and the security of the underlying hash function. In Chapter 16 we will present some variants of the RSA signature algorithm, as well as others, and show that they are secure in the sense of EUF-CMA.

15.3. The Security of RSA

In this section we examine in more detail the security of the RSA function, and the resulting "naive" encryption and signature algorithms. In particular we show how knowing the private key *is* equivalent to factoring (which should be contrasted with being able to invert the function, namely the RSA problem), how knowledge of $\phi(N)$ is also equivalent to factoring, how sharing a modulus can be a bad idea, and how also having a small public exponent could introduce problems as well.

15.3.1. Knowledge of the Private Exponent and Factoring: Whilst it is unclear whether breaking RSA, in the sense of inverting the RSA function, is equivalent to factoring, determining the private key d given the public information, N and e, is equivalent to factoring. The algorithm in the next proof is an example of a Las Vegas algorithm: It is probabilistic in nature in the sense that whilst it may not actually give an answer (or terminate), it is however guaranteed that when it does give an answer then that answer will always be correct.

Lemma 15.6. *If one knows the RSA decryption exponent d corresponding to the public key (N, e) then one can efficiently factor N.*

PROOF. Recall that for some integer s

$$e \cdot d - 1 = s \cdot (p-1) \cdot (q-1).$$

We first pick an integer $x \neq 0$, this is guaranteed to satisfy

$$x^{e \cdot d - 1} = 1 \pmod{N}.$$

We now compute a square root y_1 of one modulo N,

$$y_1 \leftarrow \sqrt{x^{e \cdot d - 1}} = x^{(e \cdot d - 1)/2},$$

which we can do since $e \cdot d - 1$ is known and will be even. We will then have the identity

$$y_1{}^2 - 1 = 0 \pmod{N},$$

which we can use to recover a factor of N via computing

$$\gcd(y_1 - 1, N).$$

But this will only work when $y_1 \neq \pm 1 \pmod{N}$.

Now suppose we are unlucky and we obtain $y_1 = \pm 1 \pmod{N}$ rather than a factor of N. If $y_1 = -1 \pmod{N}$, then we set $y_1 \leftarrow -y_1$. Thus we are always left with the case $y_1 = 1 \pmod{N}$. We take another square root of one via

$$y_2 \leftarrow \sqrt{y_1} = x^{(e \cdot d - 1)/4}.$$

Again we have

$$y_2{}^2 - 1 = y_1 - 1 = 0 \pmod{N}.$$

Hence we compute

$$\gcd(y_2 - 1, N)$$

and see whether this gives a factor of N. Again this will give a factor of N unless $y_2 = \pm 1$. If we are unlucky we repeat once more and so on.

This method can be repeated until either we have factored N or until $(e \cdot d - 1)/2^t$ is no longer divisible by 2. In this latter case we return to the beginning, choose a new random value of x and start again. □

We shall now present a small example of the previous method. Consider the following RSA parameters

$$N = 1\,441\,499, \; e = 17 \text{ and } d = 507\,905.$$

Recall that we are assuming that the private exponent d is public knowledge. We will show that the previous method does in fact find a factor of N. Put

$$t_1 \leftarrow (e \cdot d - 1)/2 = 4\,317\,192,$$
$$x \leftarrow 2.$$

To compute y_1 we evaluate

$$y_1 \leftarrow x^{(e \cdot d - 1)/2} = 2^{t_1} = 1 \pmod{N}.$$

Since we obtain $y_1 = 1$ we need to set

$$t_2 \leftarrow t_1/2 = (e \cdot d - 1)/4 = 2\,158\,596,$$
$$y_2 \leftarrow 2^{t_2}.$$

We now compute y_2,

$$y_2 \leftarrow x^{(e \cdot d - 1)/4} = 2^{t_2} = 1 \pmod{N}.$$

So we need to repeat the method again; this time we obtain $t_3 = (e \cdot d - 1)/8 = 1\,079\,298$. We compute y_3,

$$y_3 = x^{(e \cdot d - 1)/8} = 2^{t_3} = 119\,533 \pmod{N}.$$

So

$$y_3{}^2 - 1 = (y_3 - 1) \cdot (y_3 + 1) = 0 \pmod{N},$$

and we compute a prime factor of N by evaluating $\gcd(y_3 - 1, N) = 1423$.

15.3.2. Knowledge of $\phi(N)$ and Factoring: We have seen that knowledge of d allows us to factor N. Now we will show that knowledge of $\Phi = \phi(N)$ also allows us to factor N.

Lemma 15.7. *Given an RSA modulus N and the value of $\Phi = \phi(N)$ one can efficiently factor N.*

Proof. We have

$$\Phi = (p-1) \cdot (q-1) = N - (p+q) + 1.$$

Hence, if we set $S = N + 1 - \Phi$, we obtain

$$S = p + q.$$

So we need to determine p and q from their sum S and product N. Define the polynomial

$$f(X) = (X - p) \cdot (X - q) = X^2 - S \cdot X + N.$$

So we can find p and q by solving $f(X) = 0$ using the standard formulae for extracting the roots of a quadratic polynomial,

$$p = \frac{S + \sqrt{S^2 - 4 \cdot N}}{2},$$
$$q = \frac{S - \sqrt{S^2 - 4 \cdot N}}{2}.$$

□

As an example consider the RSA public modulus $N = 18\,923$. Assume that we are given $\Phi = \phi(N) = 18\,648$. We then compute

$$S = p + q = N + 1 - \Phi = 276.$$

Using this we compute the polynomial

$$f(X) = X^2 - S \cdot X + N = X^2 - 276 \cdot X + 18\,923$$

and find that its roots over the real numbers are $p = 149$, $q = 127$, which are indeed the factors of N.

15.3.3. Use of a Shared Modulus: Since modular arithmetic is very expensive it can be very tempting to set up a system in which a number of users share the same public modulus N but use different public/private exponents, (e_i, d_i). One reason to do this could be to allow very fast hardware acceleration of modular arithmetic, specially tuned to the chosen shared modulus N. This is, however, a very silly idea since it can be attacked in one of two ways, either by a malicious insider or by an external attacker.

Suppose the attacker is one of the internal users, say user number one. She can now compute the value of the decryption exponent for user number two, namely d_2. First user one computes p and q since she knows d_1, via the algorithm in the proof of Lemma 15.6. Then user one computes $\phi(N) = (p-1) \cdot (q-1)$, and finally she can recover d_2 from

$$d_2 = \frac{1}{e_2} \pmod{\phi(N)}.$$

Now suppose the attacker is not one of the people who share the modulus, and that the two public exponents e_1 and e_2 are coprime. We now present an attack against the "naive" RSA encryption algorithm in this setting. Suppose Alice sends the same message m to two of the users

with public keys (N, e_1) and (N, e_2), i.e. $N_1 = N_2 = N$. Eve, the external attacker, sees the messages c_1 and c_2, where the ciphertexts are derived by executing

$$\begin{aligned} c_1 &\leftarrow m^{e_1} \pmod N, \\ c_2 &\leftarrow m^{e_2} \pmod N. \end{aligned}$$

Eve can now compute

$$\begin{aligned} t_1 &\leftarrow {e_1}^{-1} \pmod{e_2}, \\ t_2 &\leftarrow (t_1 \cdot e_1 - 1)/e_2, \end{aligned}$$

and can recover the message m from

$$\begin{aligned} c_1^{t_1} \cdot c_2^{-t_2} &= m^{e_1 \cdot t_1} m^{-e_2 \cdot t_2} \\ &= m^{1+e_2 \cdot t_2} m^{-e_2 \cdot t_2} \\ &= m^{1+e_2 \cdot t_2 - e_2 \cdot t_2} \\ &= m^1 = m. \end{aligned}$$

As an example of this external attack, take the public keys to be

$$N = N_1 = N_2 = 18\,923, \ e_1 = 11 \text{ and } e_2 = 5.$$

Now suppose Eve sees the ciphertexts

$$c_1 = 1514 \text{ and } c_2 = 8189$$

corresponding to the same plaintext m. Then Eve computes $t_1 = 1$ and $t_2 = 2$, and recovers the message

$$m = c_1^{t_1} \cdot c_2^{-t_2} = 100 \pmod N.$$

15.3.4. Use of a Small Public Exponent: In practice RSA systems often use a small public exponent e so as to cut down the computational cost of the sender. We shall now show that this can also lead to problems for the RSA encryption algorithm. Suppose we have three users all with different public moduli N_1, N_2 and N_3. In addition suppose they all have the same small public exponent $e = 3$. Suppose someone sends them the same message m. The attacker Eve sees the messages

$$\begin{aligned} c_1 &\leftarrow m^3 \pmod{N_1}, \\ c_2 &\leftarrow m^3 \pmod{N_2}, \\ c_3 &\leftarrow m^3 \pmod{N_3}. \end{aligned}$$

Now the attacker, using the Chinese Remainder Theorem, computes the simultaneous solution to the equations

$$X = c_i \pmod{N_i} \text{ for } i = 1, 2, 3,$$

to obtain

$$X = m^3 \pmod{N_1 \cdot N_2 \cdot N_3}.$$

But since $m^3 < N_1 \cdot N_2 \cdot N_3$ we must have $X = m^3$ identically over the integers. Hence we can recover m by taking the real cube root of X.

As a simple example of this attack; take $N_1 = 323$, $N_2 = 299$ and $N_3 = 341$. Suppose Eve sees the ciphertexts

$$c_1 = 50, \ c_2 = 268 \text{ and } c_3 = 1,$$

and wants to determine the common value of m. Eve computes via the Chinese Remainder Theorem

$$X = 300\,763 \pmod{N_1 \cdot N_2 \cdot N_3}.$$

Finally, she computes over the integers $m = X^{1/3} = 67$.

This attack and the previous one are interesting since we find the message without factoring the modulus. This is, albeit slight, evidence that breaking RSA is easier than factoring. The main lesson, however, from both these attacks is that plaintext should be randomly padded before transmission. That way the same "message" is never encrypted to two different people. In addition one should probably avoid very small exponents for encryption; $e = 65\,537$ is the usual choice now in use. However, small public exponents for RSA signatures produce no such problems. So for RSA signatures it is common to see $e = 3$ being used in practice.

15.4. Some Lattice-Based Attacks on RSA

In this section we examine how lattices can be used to attack certain systems, when some other side information is known.

15.4.1. Håstad's Attack: Earlier in this chapter we saw the following attack on the RSA system: Given three public keys (N_i, e_i) all with the same encryption exponent $e_i = 3$, if a user sent the same message to all three public keys then an adversary could recover the plaintext using the Chinese Remainder Theorem. Suppose that we try to protect against this attack by insisting that before encrypting a message m we first pad with some user-specific data. For example the ciphertext becomes, for user i,

$$c_i = (i \cdot 2^h + m)^3 \pmod{N_i}.$$

However, one can still break this system using an attack due to Håstad. Håstad's attack is related to Coppersmith's Theorem since we can interpret the attack scenario, generalized to k users and public encryption exponent e, as being given k polynomials of degree e

$$g_i(x) = (i \cdot 2^h + x)^e - c_i,\ 1 \le i \le k.$$

Then given that there is an m such that

$$g_i(m) = 0 \pmod{N_i},$$

the goal is to recover m. We can assume that m is smaller than any one of the moduli N_i. Setting

$$N = N_1 \cdot N_2 \cdots N_k$$

and using the Chinese Remainder Theorem we can find t_i so that

$$g(x) = \sum_{i=1}^{k} t_i \cdot g_i(x)$$

and

$$g(m) = 0 \pmod{N}.$$

Then, since g has degree e and is monic, using Theorem 5.10 we can recover m in polynomial time, as long as we have at least as many ciphertexts/users as the encryption exponent i.e. $k \ge e$, since

$$m < \min_i N_i < N^{1/k} \le N^{1/e}.$$

15.4.2. Franklin–Reiter Attack and Coppersmith's Generalization: Now suppose we have one RSA public key (N, e) owned by Alice. The Franklin–Reiter attack applies to the following situation: Bob wishes to send two related messages m_1 and m_2 to Alice, where the relation is given by the public polynomial

$$m_1 = f(m_2) \pmod{N}.$$

We shall see that, given c_1 and c_2, an attacker has a good chance of determining m_1 and m_2 for any small encryption exponent e. The attack is particularly simple when

$$f = a \cdot x + b \text{ and } e = 3,$$

with a and b fixed and given to the attacker. The attacker knows that m_2 is a root, modulo N, of the two polynomials

$$g_1(x) = x^3 - c_2,$$
$$g_2(x) = f(x)^3 - c_1.$$

So the linear factor $x - m_2$ divides both $g_1(x)$ and $g_2(x)$.

We now form the greatest common divisor of $g_1(x)$ and $g_2(x)$. Strictly speaking this is not possible in general since $(\mathbb{Z}/N)\mathbb{Z}[x]$ is not a Euclidean ring, but if the Euclidean algorithm breaks down then we would find a factor of N and so be able to find Alice's private key in any case. One can show that when $f = a \cdot x + b$ and $e = 3$, the resulting gcd, when it exists, must always be the linear factor $x - m_2$, and so the attacker can always find m_2 and then m_1.

Coppersmith extended the attack of Franklin and Reiter in a way which also extends the padding result from Håstad's attack. Suppose before sending a message m we pad it with some random data. So for example if N is an n-bit RSA modulus and m is a k-bit message then we could append $n - k$ random bits to either the top or bottom of the message. Say

$$m' = 2^{n-k} \cdot m + r$$

where r is some, per message, random number of length $n-k$. This would seem to be a good idea in any case, since it makes the RSA function randomized, which might help in making it semantically secure. However, Coppersmith showed that this naive padding method is insecure.

Suppose Bob sends the same message to Alice twice, i.e. we have ciphertexts c_1 and c_2 corresponding to the messages

$$m_1 = 2^{n-k} \cdot m + r_1,$$
$$m_2 = 2^{n-k} \cdot m + r_2,$$

where r_1, r_2 are two different random $(n-k)$-bit numbers. The attacker sets $y_0 = r_2 - r_1$ and is led to solve the simultaneous equations

$$g_1(x, y) = x^e - c_1,$$
$$g_2(x, y) = (x + y)^e - c_2.$$

The attacker forms the resultant $h(y)$ of $g_1(x, y)$ and $g_2(x, y)$ with respect to x. Now $y_0 = r_2 - r_1$ is a small root of the polynomial $h(y)$, which has degree e^2. Using Coppersmith's Theorem 5.10 the attacker recovers $r_2 - r_1$ and then recovers m_2 using the method of the Franklin–Reiter attack.

Whilst the above trivial padding scheme is therefore insecure, one can find secure padding schemes for the RSA encryption algorithm. We shall return to padding schemes for RSA in Chapter 16.

15.4.3. Wiener's Attack on RSA: We have mentioned that often one uses a small public RSA exponent e so as to speed up the public key operations in RSA. Sometimes we have applications where it is more important to have a fast private key operation. Hence, one could be tempted to choose a small value of the private exponent d. Clearly this will lead to a large value of the encryption exponent e and we cannot choose too small a value for d, otherwise an attacker could find d using exhaustive search. However, it turns out that d needs to be at least the size of $\frac{1}{3} \cdot N^{1/4}$ due to an ingenious attack by Wiener which uses continued fractions.

Wiener's attack uses continued fractions as follows. We assume we have an RSA modulus $N = p \cdot q$ with $q < p < 2q$. In addition assume that the attacker knows that we have a small

decryption exponent $d < \frac{1}{3} \cdot N^{1/4}$. The encryption exponent e is given to the attacker, where this exponent satisfies

$$e \cdot d = 1 \pmod{\Phi},$$

with

$$\Phi = \phi(N) = (p-1) \cdot (q-1).$$

We also assume $e < \Phi$, since this holds in most systems. First notice that this means there is an integer k such that

$$e \cdot d - k \cdot \Phi = 1.$$

Hence, we have

$$\left| \frac{e}{\Phi} - \frac{k}{d} \right| = \frac{1}{d \cdot \Phi}.$$

Now, $\Phi \approx N$, since

$$|N - \Phi| = |p + q - 1| < 3 \cdot \sqrt{N}.$$

So we should have that $\frac{e}{N}$ is a close approximation to $\frac{k}{d}$.

$$\begin{aligned}
\left| \frac{e}{N} - \frac{k}{d} \right| &= \left| \frac{e \cdot d - N \cdot k}{d \cdot N} \right| \\
&= \left| \frac{e \cdot d - k \cdot \Phi - N \cdot k + k \cdot \Phi}{d \cdot N} \right| \\
&= \left| \frac{1 - k \cdot (N - \Phi)}{d \cdot N} \right| \\
&\leq \left| \frac{3 \cdot k \cdot \sqrt{N}}{d \cdot N} \right| \\
&= \frac{3 \cdot k}{d \cdot \sqrt{N}}.
\end{aligned}$$

Since $e < \Phi$, it is clear that we have $k < d$, which is itself less than $\frac{1}{3} \cdot N^{1/4}$ by assumption. Hence,

$$\left| \frac{e}{N} - \frac{k}{d} \right| < \frac{1}{2 \cdot d^2}.$$

Since $\gcd(k, d) = 1$ we see that $\frac{k}{d}$ will be a fraction in its lowest terms. Hence, the fraction

$$\frac{k}{d}$$

must arise as one of the convergents of the continued fraction expansion of

$$\frac{e}{N}.$$

The correct one can be detected by simply testing which one gives a d which satisfies

$$(m^e)^d = m \pmod{N}$$

for some random value of m. The total number of convergents we will need to take is of order $O(\log N)$, hence the above gives a linear-time algorithm to determine the private exponent when it is less than $\frac{1}{3} \cdot N^{1/4}$.

As an example suppose we have the RSA modulus

$$N = 9\,449\,868\,410\,449$$

with the public key

$$e = 6\,792\,605\,526\,025.$$

We are told that the decryption exponent satisfies $d < \frac{1}{3} \cdot N^{1/4} \approx 584$. To apply Wiener's attack we compute the continued fraction expansion of the number

$$\alpha = \frac{e}{N},$$

and check each denominator of a convergent to see whether it is equal to the private key d. The convergents of the continued fraction expansion of α are given by

$$1, \frac{2}{3}, \frac{3}{4}, \frac{5}{7}, \frac{18}{25}, \frac{23}{32}, \frac{409}{569}, \frac{1659}{2308}, \ldots$$

Checking each denominator in turn we see that the decryption exponent is given by

$$d = 569,$$

which is the denominator of the seventh convergent.

15.4.4. Extension to Wiener's Attack: Boneh and Durfee, using an analogue of the bivariate case of Coppersmith's Theorem 5.10, extended Wiener's attack to the case where

$$d \leq N^{0.292},$$

using a heuristic algorithm, i.e. the range of "bad" values of d was extended further. We do not go into the details but show how Boneh and Durfee proceed to a problem known as the small inverse problem. Suppose we have an RSA modulus N, with encryption exponent e and decryption exponent d. By definition there is an integer k such that

$$e \cdot d + \frac{k \cdot \Phi}{2} = 1,$$

where $\Phi = \phi(N)$. Expanding the definition of Φ we find

$$e \cdot d + k \cdot \left(\frac{N+1}{2} - \frac{p+q}{2} \right) = 1.$$

We set

$$s = -\frac{p+q}{2},$$
$$A = \frac{N+1}{2}.$$

Then finding d, where d is small, say $d < N^{\delta}$, is equivalent to finding the two small solutions k and s to the following congruence

$$f(k, s) = k \cdot (A + s) = 1 \pmod{e}.$$

To see that k and s are small relative to the modulus e for the above equation, notice that $e \approx N$ since d is small, and so

$$|s| < 2 \cdot N^{0.5} \approx e^{0.5} \text{ and } |k| < \frac{2 \cdot d \cdot e}{\Phi} \leq \frac{3 \cdot d \cdot e}{N} \approx e^{\delta}.$$

We can interpret this problem as finding an integer which is close to A whose inverse is small modulo e. This is called the small inverse problem. Boneh and Durfee show that this problem has a solution when $\delta \leq 0.292$, hence extending Wiener's attack. This is done by applying the multivariate analogue of Coppersmith's method to the polynomial $f(k, s)$.

15.5. Partial Key Exposure Attacks on RSA

Partial key exposure is related to the following question: Suppose in some cryptographic scheme the attacker recovers a certain set of bits of the private key, can the attacker use this to recover the whole private key? In other words, does partial exposure of the key result in a total break of the system? We shall present a number of RSA examples, however these are not the only ones. There are a number of results related to partial key exposure which relate to other schemes such as DSA or symmetric-key-based systems.

15.5.1. Partial Exposure of the MSBs of the RSA Decryption Exponent: Somewhat surprisingly for RSA, in the more common case of using a small public exponent e, one can trivially recover half of the bits of the private key d, namely the most significant ones, as follows. Recall that there is a value of k such that $0 < k < e$ with

$$e \cdot d - k \cdot (N - (p + q) + 1) = 1.$$

Now suppose for each possible value of i, $0 < i \leq e$, the attacker computes

$$d_i = \lfloor (i \cdot N + 1)/e \rfloor.$$

Then we have

$$|d_k - d| \leq k \cdot (p + q)/e \leq 3 \cdot k \cdot \sqrt{N}/e < 3 \cdot \sqrt{N}.$$

Hence, d_k is a good approximation for the actual value of d.

Now when $e = 3$ it is clear that with high probability we have $k = 2$ and so d_2 reveals half of the most significant bits of d. Unluckily for the attack, and luckily for the user, there is no known way to recover the rest of d given only the most significant bits.

15.5.2. Partial Exposure of Some Bits of the RSA Prime Factors: Suppose our n-bit RSA modulus N is given by $p \cdot q$, with $p \approx q$, and that the attacker has found the $n/4$ least significant bits of p. Recall that p is only around $n/2$ bits long in any case, so this means the attacker is given the lower half of all the bits making up p. We write

$$p = x_0 \cdot 2^{n/4} + p_0.$$

We then have, writing $q = y_0 \cdot 2^{n/4} + q_0$,

$$N = p_0 \cdot q_0 \pmod{2^{n/4}}.$$

Hence, we can determine the value of q_0. We now write down the polynomial

$$\begin{aligned} p(x, y) &= (p_0 + 2^{n/4} \cdot x) \cdot (q_0 + 2^{n/4} \cdot y) \\ &= p_0 \cdot q_0 + 2^{n/4} \cdot (p_0 \cdot y + q_0 \cdot x) + 2^{n/2} \cdot x \cdot y. \end{aligned}$$

Now $p(x, y)$ is a bivariate polynomial of degree two which has known small solution modulo N, namely (x_0, y_0) where $0 < x_0, y_0 \leq 2^{n/4} \approx N^{1/4}$. Hence, using the heuristic bivariate extension of Coppersmith's Theorem 5.10, we can recover x_0 and y_0 in polynomial time and so factor the modulus N. A similar attack applies when the attacker knows the $n/4$ most significant bits of p.

15.5.3. Partial Exposure of the LSBs of the RSA Decryption Exponent: We now suppose we are given, for small public exponent e, a quarter of the least significant bits of the private exponent d. That is we have d_0 where

$$d = d_0 + 2^{n/4} \cdot x_0$$

where $0 \leq x_0 \leq 2^{3 \cdot n/4}$. Recall that there is a value of k with $0 < k < e$ such that

$$e \cdot d - k \cdot (N - (p + q) + 1) = 1.$$

We then have, since $N = p \cdot q$,

$$e \cdot d \cdot p - k \cdot p \cdot (N - p + 1) + k \cdot N = p.$$

If we set $p_0 = p \pmod{2^{n/4}}$ then we have the equation

$$e \cdot d_0 \cdot p_0 - k \cdot p_0 \cdot (N - p_0 + 1) + k \cdot N - p_0 = 0 \pmod{2^{n/4}}. \tag{22}$$

This gives us the following algorithm to recover the whole of d. For each value of k less than e we solve equation (22) modulo $2^{n/4}$ for p_0. Each value of k will result in $O(\frac{n}{4})$ possible values for p_0. Using each of these values of p_0 in turn, we apply the previous technique for factoring N from Section 15.5.2. One such value of p_0 will be the correct value of $p \pmod{2^{n/4}}$ and so the above factorization algorithm will work and we can recover the value of d.

15.6. Fault Analysis

An interesting class of attacks results from trying to induce faults within a cryptographic system. We shall describe this area in relation to the RSA signature algorithm but similar attacks can be mounted on other cryptographic algorithms, both public and symmetric key. Imagine we have a hardware implementation of RSA, in a smart card say. On input of some message m the chip will sign the message for us, using some internal RSA private key. The attacker wishes to determine the private key hidden within the smart card. To do this the attacker can try to make the card perform some of the calculation incorrectly, by either altering the card's environment by heating or cooling it or by damaging the circuitry of the card in some way.

An interesting case is when the card uses the Chinese Remainder Theorem to perform the signing operation, to increase efficiency, as explained in Chapter 6. The card first computes the hash of the message

$$h = H(m).$$

Then the card computes

$$s_p = h^{d_p} \pmod{p},$$
$$s_q = h^{d_q} \pmod{q},$$

where $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$. The final signature is produced by the card from s_p and s_q via the Chinese Remainder Theorem using

- $u = (s_q - s_p) \cdot T \pmod{q}$,
- $s = s_p + u \cdot p$,

where $T = p^{-1} \pmod{q}$.

Now suppose that the attacker can introduce a fault into the computation so that s_p is computed incorrectly. The attacker will then obtain a value of s such that

$$s^e \neq h \pmod{p},$$
$$s^e = h \pmod{q}.$$

Hence, by computing

$$q = \gcd(s^e - h, N)$$

she can factor the modulus.

Chapter Summary

- RSA is the most popular public key encryption algorithm, but its security rests on the difficulty of the RSA problem and not quite on the difficulty of FACTOR.

- Rabin encryption is based on the difficulty of extracting square roots modulo a composite modulus. Since the problems SQRROOT and FACTOR are polynomial-time equivalent this means that Rabin encryption is based on the difficulty of FACTOR.
- Naive RSA is not IND-CPA secure.
- The RSA encryption algorithm can be used in reverse to produce a public key signature scheme, but one needs to combine the RSA algorithm with a hash algorithm to obtain security for both short and long messages.
- Using a small private decryption exponent in RSA is not a good idea due to Wiener's attack.
- The Håstad and Franklin–Reiter attacks imply that one needs to be careful when designing padding schemes for RSA.
- Using Coppersmith's Theorem one can show that revealing a proportion of the bits of either p, q or d in RSA can lead to a complete break of the system.

Further Reading

Still the best quick introduction to the concept of public key cryptography can be found in the original paper of Diffie and Hellman; see also the original paper on RSA encryption. Our treatment of attacks on RSA in this chapter has closely followed the survey article by Boneh.

D. Boneh. *Twenty years of attacks on the RSA cryptosystem.* Notices of the American Mathematical Society (AMS), **46**, 203–213, 1999.

W. Diffie and M. Hellman. *New directions in cryptography.* IEEE Trans. on Info. Theory, **22**, 644–654, 1976.

R.L. Rivest, A. Shamir and L.M. Adleman. *A method for obtaining digital signatures and public-key cryptosystems.* Comm. ACM, **21**, 120–126, 1978.

CHAPTER 16

Public Key Encryption and Signature Algorithms

Chapter Goals

- To present fully secure public key encryption schemes and signature schemes.
- To show how the random oracle model can be used to prove certain encryption and signature schemes are secure.
- To understand how the RSA algorithm is actually used in practice, and sketch a proof of RSA-OAEP.
- To introduce and formalize the notion of hybrid encryption, via KEMs and DEMs.
- To present two efficient KEMs, namely RSA-KEM and DHIES-KEM.
- To explain the most widely used signature algorithms, namely variants of RSA and DSA.
- To present the Cramer–Shoup encryption and signature schemes, which do not require the use of the random oracle model.

16.1. Passively Secure Public Key Encryption Schemes

In this section we present three basic passively secure encryption schemes, namely the Goldwasser–Micali encryption scheme, the ElGamal encryption scheme, and the Paillier encryption scheme.

16.1.1. Goldwasser–Micali Encryption: We have seen that RSA is not semantically secure even against a passive attack; thus it would be nice to give a system which is IND-CPA secure and is based on some factoring-like assumption. Historically the first system to meet these goals was one by Goldwasser and Micali. The scheme is not used in real-life applications, due to its inefficiency, however its simplicity means that it can help solidify ideas about to how construct (and prove secure) the systems we do use in real life.

The security of the Goldwasser–Micali encryption scheme is based on the hardness of the QUADRES problem, namely given a composite integer N and an integer e, it is hard to test whether a is a quadratic residue or not without knowledge of the factors of N. Let us recap, from Chapter 2, that the set of squares in $(\mathbb{Z}/N\mathbb{Z})^*$ is denoted by

$$Q_N = \{x^2 \pmod N : x \in (\mathbb{Z}/N\mathbb{Z})^*\},$$

and J_N denotes the set of elements with Jacobi symbol equal to plus one, i.e.

$$J_N = \left\{a \in (\mathbb{Z}/N\mathbb{Z})^* : \left(\frac{a}{N}\right) = 1\right\}.$$

The set of pseudo-squares is the difference $J_N \setminus Q_N$. For an RSA-like modulus $N = p \cdot q$ the number of elements in J_N is equal to $(p-1)\cdot(q-1)/2$, whilst the number of elements in Q_N is $(p-1)\cdot(q-1)/4$. The QUADRES problem is that given an element x of J_N, it is hard to tell whether $x \in Q_N$, whilst it is easy to tell whether $x \in J_N$ or not.

We can now explain the Goldwasser–Micali encryption system.

Key Generation: As a private key we take two large prime numbers $\mathfrak{sk} = (p, q)$ and then compute the public modulus $N \leftarrow p \cdot q$, and an integer $y \in J_N \setminus Q_N$. The public key is set to be $\mathfrak{pk} \leftarrow (N, y)$. The value of y is computed by the public key owner by first computing elements $y_p \in \mathbb{F}_p^*$ and $y_q \in \mathbb{F}_q^*$ such that

$$\left(\frac{y_p}{p}\right) = \left(\frac{y_q}{q}\right) = -1.$$

Then the value of y is computed from y_p and y_q via the Chinese Remainder Theorem. A value of y computed in this way clearly does not lie in Q_N, but it does lie in J_N since

$$\left(\frac{y}{N}\right) = \left(\frac{y}{p}\right) \cdot \left(\frac{y}{q}\right) = \left(\frac{y_p}{p}\right) \cdot \left(\frac{y_q}{q}\right) = (-1) \cdot (-1) = 1.$$

Encryption: The Goldwasser–Micali encryption system encrypts one bit of information at a time. To encrypt the bit b,

- $x \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$.
- $c \leftarrow y^b \cdot x^2 \pmod{N}$.

The ciphertext is then the value of c. Notice that this is very inefficient since a single bit of plaintext requires $\log_2 N$ bits of ciphertext to transmit it.

Decryption: Notice that the ciphertext c will always be an element of J_N. However, if the message bit b is zero then the value of c will be a quadratic residue, otherwise it will be a quadratic non-residue. So all the decryptor has to do to recover the message is determine whether c is a quadratic residue or not modulo N. But the decryptor is assumed to know the factors of N and so can compute the Legendre symbol

$$\left(\frac{c}{p}\right).$$

If this Legendre symbol is equal to plus one then c is a quadratic residue and so the message bit is zero. If however the Legendre symbol is equal to minus one then c is not a quadratic residue and so the message bit is one.

It is now relatively straightforward to prove that the Goldwasser–Micali encryption scheme is IND-CPA secure, assuming that the QUADRES problem is hard for RSA style moduli N of size v bits.

Theorem 16.1. *Suppose there is an adversary A against the* IND-CPA *security of the Goldwasser–Micali encryption scheme Π for moduli of size v bits, then there is an adversary B against the QUADRES problem such that*

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CPA}}(A) = 2 \cdot \mathrm{Adv}_{v}^{\mathsf{QUADRES}}(B).$$

Proof. We describe the algorithm B which will use A as an oracle. To see this in pictures see Figure 16.1. Suppose algorithm B is given N and $j \in J_N$ and is asked to determine whether $j \in Q_N$. Algorithm B first randomizes j to form y, on the assumption that j *does not* lie in Q_N. Thus algorithm B sets $y \leftarrow j \cdot z^2 \pmod{N}$, for some $z \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$. The public key $\mathfrak{pk} \leftarrow (N, y)$ is then passed to algorithm A.

Since the Goldwasser–Micali system only encrypts bits we can assume that the **find** stage of the adversary A will simply output the two messages

$$m_0 = 0 \text{ and } m_1 = 1.$$

We now form the challenge ciphertext

$$c^* \leftarrow y^b \cdot r^2,$$

for some bit $b \leftarrow \{0,1\}$ and some random $r \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$ chosen by algorithm B. We now pass c^* to algorithm A, which will respond with its guess b' for the bit b. If $b = b'$ then algorithm B returns that j is a quadratic residue, otherwise it returns that it is not.

To analyse the probabilities we notice that if j is not a quadratic residue then this value of c^* will be a valid encryption of the message m_b. So if j is a quadratic residue then algorithm B is presenting a valid challenger to algorithm A. However, if j is not a quadratic residue then this is not a valid encryption of anything (since the public key is not even valid). Thus we have

$$\begin{aligned}\mathrm{Adv}_v^{\mathsf{QUADRES}}(B) &= \left|\Pr[b' = b | y \in Q_N] - \Pr[b' = b | y \in J_N \setminus Q_N]\right| \\ &= \left|\Pr[A \text{ wins for a valid challenger}] - \frac{1}{2}\right| \\ &= \frac{1}{2} \cdot \mathrm{Adv}_\Pi^{\mathsf{IND\text{-}CPA}}(A)\end{aligned}$$

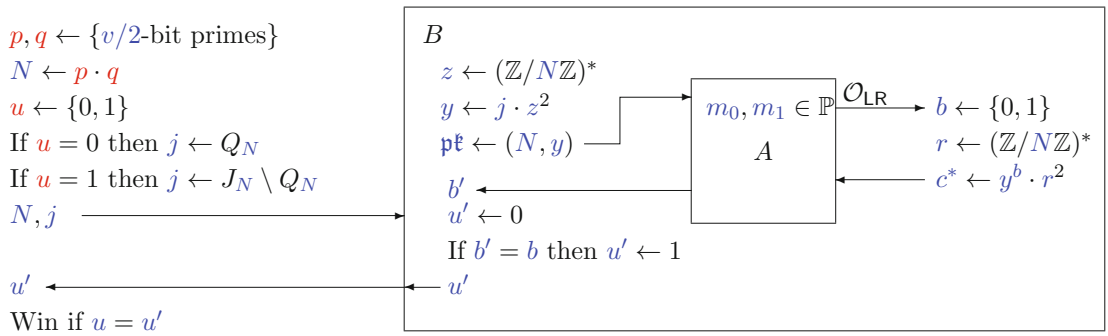


FIGURE 16.1. How B interacts with A in Theorem 16.1

□

Note that the above argument says nothing about whether the Goldwasser–Micali encryption scheme is secure against adaptive adversaries. In fact, one can show it is not secure against such adversaries.

Theorem 16.2. *The Goldwasser–Micali encryption scheme is not* IND-CCA *secure.*

PROOF. Suppose c^* is the target ciphertext and we want to determine what bit b is encrypted by c^*. Recall that $c^* = y^b \cdot x^2 \pmod N$. Now the rules of the game do not allow us to ask our decryption oracle to decrypt c^*, but we can ask our oracle to decrypt any other ciphertext. We therefore produce the ciphertext

$$c = c^* \cdot z^2 \pmod N,$$

for some random value $z \in (\mathbb{Z}/N\mathbb{Z})^*$. It is easy to see that c is an encryption of the same bit b. Hence, by asking our oracle to decrypt c we will obtain the decryption of c^*. □

16.1.2. ElGamal Encryption: The Goldwasser–Micali encryption scheme is passively secure, but not efficient. What we really want is a simple encryption algorithm which is efficient and which is passively secure[1]. The simplest efficient IND-CPA secure encryption algorithm is the ElGamal encryption algorithm, which is based on the discrete logarithm problem. In the following we shall

[1]At this point we still focus on passively secure systems. Once we have solved this, we will turn our focus to actively secure schemes.

describe the finite field analogue of ElGamal encryption; we leave it as an exercise to write down the elliptic curve variant.

Domain Parameters: Unlike the RSA algorithm, in ElGamal encryption there are some public parameters which can be shared by a number of users. These are called the domain parameters and are given by

- p a "large prime", by which we mean one with around 2048 bits, such that $p-1$ is divisible by another "medium prime" q of around 256 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e. $g = r^{(p-1)/q} \pmod{p} \neq 1$ for some $r \in \mathbb{F}_p^*$.

The domain parameters create a public finite abelian group G of prime order q with generator g.

Key Generation: Once these domain parameters have been fixed, the public and private keys can then be determined. The private key $\mathfrak{sk}$ is chosen to be an integer $x \leftarrow [0, \ldots, q-1]$, whilst the public key is given by $\mathfrak{pk} := h \leftarrow g^x \pmod{p}$. Notice that, whilst each user in RSA needed to generate two large primes to set up their key pair (which is a costly task), for ElGamal encryption each user only needs to generate a random number and perform a modular exponentiation to generate a key pair.

Encryption: Messages are assumed to be elements of the group G. To encrypt a message $m \in G$ we do the following:

- $k \leftarrow \{0, \ldots, q-1\}$
- $c_1 \leftarrow g^k$,
- $c_2 \leftarrow m \cdot h^k$,
- Output the ciphertext, $c \leftarrow (c_1, c_2) \in G \times G$.

Notice that since each message has a different ephemeral key k, encrypting the same message twice will produce different ciphertexts.

Decryption: To decrypt a ciphertext $c = (c_1, c_2)$ we compute

$$\frac{c_2}{{c_1}^x} = \frac{m \cdot h^k}{g^{x \cdot k}} = \frac{m \cdot g^{x \cdot k}}{g^{x \cdot k}} = m.$$

ElGamal Example: We first need to set up the domain parameters. For our small example we choose $q = 101$, $p = 809$ and $g = 256$. Note that q divides $p-1$ and that g has order q, in the multiplicative group of integers modulo p. As a public/private key pair we choose

- $x \leftarrow 68$,
- $h \leftarrow g^x = 498$.

Now suppose we wish to encrypt the message $m = 100$ to the user with the above ElGamal public key.

- We generate a random ephemeral key $k \leftarrow 89$.
- Set $c_1 \leftarrow g^k = 468$.
- Set $c_2 \leftarrow m \cdot h^k = 494$.
- Output the ciphertext as $c = (468, 494)$.

The recipient can decrypt our ciphertext by computing

$$\frac{c_2}{{c_1}^x} = \frac{494}{468^{68}} = 100.$$

This last value is computed by first computing 468^{68}, taking the inverse modulo p of the result and then multiplying this value by 494.

We can now start to establish basic security results about ElGamal encryption by presenting two results in the passive security setting. Our first one says that if the Diffie–Hellman problem

is hard then ElGamal is OW-CPA, whilst the second one says that if the Decision Diffie–Hellman problem is hard then ElGamal is IND-CPA.

Theorem 16.3. *If A is an adversary against the* OW-CPA *security of the ElGamal encryption scheme $\Pi(G)$ over the group G, then there is an adversary B against the Diffie–Hellman problem such that*

$$\mathrm{Adv}^{\mathsf{OW\text{-}CPA}}_{\Pi(G)}(A) = \mathrm{Adv}^{\mathsf{DHP}}_{G}(B).$$

Proof. The algorithm A takes as input a public key h and a target ciphertext $c^* = (c_1, c_2)$, and returns the underlying plaintext. We will show how to use this algorithm to create an algorithm B to solve the DHP. We suppose B is given $X = g^x$ and $Y = g^y$, and is asked to solve the Diffie–Hellman problem, i.e. to output the value of $g^{x\cdot y}$; algorithm B then proceeds as in Algorithm 16.1.

Algorithm 16.1: Algorithm to solve DHP given an algorithm to break the one-way security of ElGamal

As input we have $X = g^x \in G$ and $Y = g^y \in G$.
$h \leftarrow X = g^x$.
$c_1 \leftarrow Y = g^y$.
$c_2 \leftarrow G$.
$c^* \leftarrow (c_1, c_2)$.
$m \leftarrow A(c^*, h)$.
return c_2/m.

In words, algorithm B first sets up an ElGamal public key which depends on the input to the Diffie–Hellman problem, i.e. we set $h \leftarrow X = g^x$ (note that algorithm B does not know what the corresponding private key is). Now we write down the target "ciphertext" $c^* = (c_1, c_2)$, where

- $c_1 \leftarrow Y = g^y$,
- $c_2 \leftarrow G$, i.e. a random element of the group.

This ciphertext is sent to algorithm A, along with the public key h. Algorithm A will then output (if successful) the underlying plaintext, We then solve the original Diffie–Hellman problem by computing

$$Z \leftarrow \frac{c_2}{m} = \frac{m \cdot h^y}{m} = h^y = g^{x\cdot y}.$$

□

We can use a similar technique to prove that ElGamal is IND-CPA, but now we have to assume that the Decision Diffie–Hellman problem is hard. Notice that to obtain a stronger notion of security, we have to assume a weaker problem is hard, i.e. make a stronger assumption.

Theorem 16.4. *If A is an adversary against the* IND-CPA *security of the ElGamal encryption scheme $\Pi(G)$ over the group G, then there is an adversary B against the Decision Diffie–Hellman problem such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi(G)}(A) = 2 \cdot \mathrm{Adv}^{\mathsf{DDH}}_{G}(B).$$

Proof. As usual we will use algorithm A as a subroutine called by algorithm B. Our algorithm B for solving now proceeds as in Algorithm 16.2; to see why this algorithm solves the DDH problem consider the following argument.

- In the case when $z = x \cdot y$ then the encryption input into the guess stage of algorithm A will be a valid encryption of m_b. Hence, if algorithm A can really break the semantic security of ElGamal encryption then the output b' will be correct and the algorithm will return **true**.

Algorithm 16.2: Algorithm to solve DDH given an algorithm to break the semantic security of ElGamal

As input we have $X = g^x, Y = g^y$ and $Z = g^z$.
$h \leftarrow X = g^x$.
$(m_0, m_1, s) \leftarrow A(\textbf{find}, h)$.
$c_1 \leftarrow Y = g^y$.
$b \leftarrow \{0, 1\}$.
$c_2 \leftarrow m_b \cdot g^z$.
$c^* \leftarrow (c_1, c_2)$.
$b' \leftarrow A(\textbf{guess}, c^*, s)$.
if $b = b'$ **then return true**.
else return false.

- Now suppose that $z \neq x \cdot y$, then the encryption input into the guess stage is almost definitely invalid, i.e. not an encryption of m_1 or m_2. Hence, the output b' of the guess stage will be independent of the value of b. Therefore we expect the above algorithm to return **true** or **false** with equal probability, and so $b' = b$ with probability $1/2$.

This is exactly the same argument that we had in the proof of security of the Goldwasser–Micali encryption scheme, and so the relationship between the advantages will follow in the same way. □

Despite the above positive results on the security of ElGamal encryption we still do not have a scheme which is secure against adaptive chosen ciphertext attacks. The main reason for this is that ElGamal is trivially malleable. Given a ciphertext for the message m,

$$(c_1, c_2) = (g^k, m \cdot h^k),$$

one can then create a valid ciphertext for the message $2 \cdot m$ without ever knowing m, nor the ephemeral key k, nor the private key x. In particular the following ciphertext decrypts to $2 \cdot m$,

$$(c_1, 2 \cdot c_2) = (g^k, 2 \cdot m \cdot h^k).$$

One can use this malleability property, just as we did with RSA in Lemma 15.4, to show that ElGamal encryption is not OW-CCA secure. Notice that $(1, 2)$ is a "trivial" encryption of the number 2, and we are in some sense combining the ciphertext $(1, 2)$ with the ciphertext (c_1, c_2) to produce a ciphertext which encrypts $2 \cdot m$. Any two ciphertexts can be combined in this way, to produce a ciphertext which encrypts the product of the underlying plaintexts. An encryption scheme with this property is called *multiplicatively homomorphic*.

Lemma 16.5. *ElGamal is not* OW-CCA.

Proof. Suppose the message the adversary wants to invert is $c^* = (c_1, c_2) = (g^k, m^* \cdot h^k)$. The adversary then creates the related message $c = (c_1, 2 \cdot c_2)$ and asks her decryption oracle to decrypt c to give the message m. Then Eve computes

$$\frac{m}{2} = \frac{2 \cdot c_2 \cdot {c_1}^{-x}}{2} = \frac{2 \cdot m^* \cdot h^k \cdot g^{-x \cdot k}}{2} = \frac{2 \cdot m^* \cdot g^{x \cdot k} \cdot g^{-x \cdot k}}{2} = \frac{2 \cdot m^*}{2} = m^*.$$

□

16.1.3. Paillier Encryption: There is an efficient system, due to Paillier, based on the difficulty of factoring large integers, which can be shown to be IND-CPA. Paillier's scheme has a number of interesting properties, such as the fact that it is additively homomorphic (which means it has found application in electronic voting applications).

Key Generation: We first pick an RSA modulus $N = p \cdot q$, but instead of working with the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$ we work with $(\mathbb{Z}/N^2\mathbb{Z})^*$. The order of this last group is given by $\phi(N) = N \cdot (p-1) \cdot (q-1) = N \cdot \phi(N)$. This means, by Lagrange's Theorem, that for all a with $\gcd(a, N) = 1$ we have

$$a^{N \cdot (p-1) \cdot (q-1)} = 1 \pmod{N^2}.$$

The private key for Paillier's scheme is defined to be an integer d such that

$$d = 1 \pmod{N},$$
$$d = 0 \pmod{(p-1) \cdot (q-1)},$$

such a value of d can be found by the Chinese Remainder Theorem. The public key $\mathfrak{pk}$ is just the integer N, and the private key $\mathfrak{sk}$ is the integer d.

Encryption: Messages are defined to be elements of $\mathbb{Z}/N\mathbb{Z}$. To encrypt such a message the encryptor picks an integer $r \in \mathbb{Z}/N^2\mathbb{Z}$ and computes $c \leftarrow (1+N)^m \cdot r^N \pmod{N^2}$.

Decryption: To decrypt one first computes

$$\begin{aligned} t &\leftarrow c^d \pmod{N^2} \\ &= (1+N)^{m \cdot d} \cdot r^{d \cdot N} \pmod{N^2} \\ &= (1+N)^{m \cdot d} \pmod{N^2} && \text{since } d = 0 \pmod{(p-1) \cdot (q-1)} \\ &= 1 + m \cdot d \cdot N \pmod{N^2} \\ &= 1 + m \cdot N \pmod{N^2} && \text{since } d = 1 \pmod{N}. \end{aligned}$$

Then to recover the message we compute $R \leftarrow \frac{t-1}{N}$.

Just like the other schemes presented so far, Paillier encryption is malleable, and hence it cannot be OW-CCA secure. However, unlike RSA, Rabin and ElGamal, the malleability is *additive*. In particular given two ciphertexts, $c_1 = (1+N)^{m_1} \cdot {r_1}^N$ and $c_2 == (1+N)^{m_2} \cdot {r_2}^N$, encrypting messages m_1 and m_2 we can easily form the encryption of the sum of the plaintexts by computing

$$c_1 \cdot c_2 = \left((1+N)^{m_1} \cdot {r_1}^N\right) \cdot \left((1+N)^{m_2} \cdot {r_2}^N\right) = (1+N)^{m_1+m_2} \cdot (r_1 \cdot r_2)^N.$$

Thus we say that Paillier encryption is *additively homomorphic*; this should be compared to the *multiplicatively homomorphic* nature of RSA, ElGamal encryption and Rabin encryption. Finding an IND-CPA secure encryption scheme which is simultaneously both additively and multiplicatively homomorphic was a major open research question in cryptography for over thirty years. Such a Fully Homomorphic Encryption (FHE) scheme, was given in 2009 by Gentry, and we shall return to such FHE schemes in Chapter 17.

The Paillier encryption scheme can be proved to be IND-CPA secure assuming the following generalization of the QUADRES problem is secure. Instead of detecting whether something is a square modulo N, the adversary needs to detect whether something is an Nth power modulo N^2. We present the problem diagrammatically in Figure 16.2, and leave it to the reader to show that Paillier encryption is IND-CPA under this assumption.

16.2. Random Oracle Model, OAEP and the Fujisaki–Okamoto Transform

We have now seen various proofs of security for public key encryption schemes, yet none of them prove security against adversaries which can make chosen ciphertext queries. Let us see why this might be a problem for our existing proof techniques. To recap, we are given an algorithm A and the proof proceeds by trying to create a new algorithm B which uses A as a subroutine. The input to B is the hard mathematical problem we wish to solve (e.g. factoring), whilst the input to A is some cryptographic problem. Since we have a public key algorithm adversary A can make its own

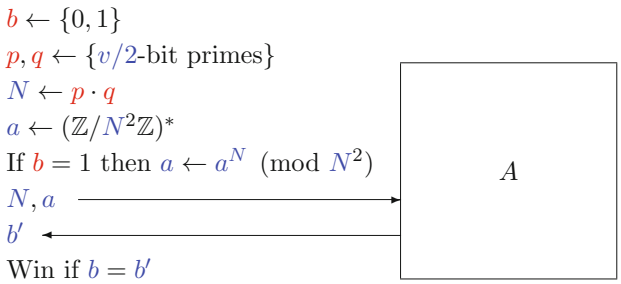


FIGURE 16.2. Security game to define the Decision Composite Residuosity Problem (DCRP)

encryption queries, as soon as it is given the public key. Thus when A is a CPA adversary there are no oracle queries which have to be answered by B. The usual trick in these proofs is that the hard mathematical problem is based on some secret which B does not know, and that this secret becomes the secret key of the public key cryptosystem which A is trying to break.

The difficulty arises when A is a CCA adversary; in this case A is allowed to call a decryption oracle for the input public key. The algorithm B, if it wants to use algorithm A as a subroutine, needs to supply the answers to A's oracle queries. In constructing algorithm B we now have a number of problems:

- Its responses must appear valid (i.e. valid encryptions should decrypt), otherwise algorithm A would notice its decryption oracle was lying. Hence, algorithm B could no longer guarantee that algorithm A was successful with non-negligible probability.
- The responses of the decryption oracle should be consistent with the probability distributions of responses that A expects if the oracle was a true decryption oracle. Again, otherwise A would notice.
- The responses of the decryption oracle should be consistent across all the calls made by the adversary A.
- Algorithm B needs to supply these answers without knowing the secret key. To decrypt we appear to need to know the secret key, which is exactly what B does not have. In most cases if B knew the secret key it would not need A in the first place!

This last point is the most crucial one. We are essentially asking B to decrypt a ciphertext without knowing the private key, but this is meant to be impossible since our scheme is meant to be secure.

To get around this problem it has become common practice to use the "random oracle model", which we introduced in Chapter 11. Recall that a random oracle is an idealized hash function which on input of a new query will pick, uniformly at random, some response from its output domain, and which if asked the same query twice will always return the same response. So to use the random oracle model we need to include a hash function somewhere in the processing of our encryption and/or decryption operations.

In the random oracle model we assume our adversary A makes no use of the explicit hash function being used in the scheme under attack. In other words the adversary A runs, and is successful, even if we replace the real hash function by a random oracle. The algorithm B responds to the decryption oracle queries of A by cheating and "cooking" the responses of the random oracle to suit his own needs.

A proof in the random oracle model is an even more relativized proof than that which we considered before. Such a proof says that assuming some problem is hard, say factoring, then an adversary cannot exist which makes no use of the underlying hash function. This does not imply that an adversary does not exist which uses the real specific hash function as a means of breaking the cryptographic system.

16.2.1. RSA-OAEP: Recall that the raw RSA function does not provide a semantically secure encryption scheme, even against passive adversaries. To make a system which is secure we need either to add redundancy to the plaintext before encryption or to add some other form of redundancy to the ciphertext, so that we can check upon decryption whether the ciphertext has been validly generated. In addition the padding used needs to be random so as to make a non-deterministic encryption algorithm. Over the years a number of padding systems have been proposed. However, many of the older ones are now considered weak.

By far the most successful padding scheme in use today was invented by Bellare and Rogaway and is called OAEP or Optimized Asymmetric Encryption Padding. The general OAEP method is a padding scheme which can be used with any function which is a trapdoor one-way permutation on strings of k bits in length. When used with the RSA trapdoor one-way permutation we need to "tweak" the construction a little since RSA does not act as a permutation on bit strings of length k, it acts as a permutation on the set of integers modulo N. When used with RSA it is often denoted RSA-OAEP.

Originally it was thought that OAEP was a plaintext aware encryption algorithm in the random oracle model, irrespective of the underlying trapdoor one-way permutation, but this claim has since been shown to be wrong. However, one can show in the random oracle model that RSA-OAEP is semantically secure against adaptive chosen ciphertext attacks.

We first give the description of OAEP in general. Let f be any k-bit to k-bit trapdoor one-way permutation. Let k_0 and k_1 denote numbers such that a work effort of 2^{k_0} or 2^{k_1} is impossible (e.g. $k_0, k_1 > 128$). Put $n = k - k_0 - k_1$ and let

$$G : \{0,1\}^{k_0} \longrightarrow \{0,1\}^{n+k_1}$$
$$H : \{0,1\}^{n+k_1} \longrightarrow \{0,1\}^{k_0}$$

be hash functions. Strictly speaking H *is* a hash function as it takes bitstrings and compresses them in length, whereas G is a function more like a key derivation function in that it expands a short bit string into a longer one. In practice for RSA-OAEP both F and G are implemented using hash functions, with G being implemented by repeated hashing of the input along with a counter as in Section 14.6.

Let m be a message of n bits in length. We then encrypt using the function

$$c \leftarrow E(m) = f\left(\{\left(m \parallel 0^{k_1}\right) \oplus G(R)\} \parallel \{R \oplus H\left(\left(m \| 0^{k_1}\right) \oplus G(R)\right)\}\right) = f(A).$$

where

- $m \parallel 0^{k_1}$ means m followed by k_1 zero bits,
- R is a random bit string of length k_0,
- $\parallel$ denotes concatenation.

One can view OAEP as a two-stage Feistel network, as Figure 16.3 demonstrates. To decrypt we proceed as follows:

- Apply the trapdoor to f to recover $A = f^{-1}(c) = \{T \| \{R \oplus H(T)\}\}$.
- Compute $H(T)$ and recover R from $R \oplus H(T)$.
- Compute $G(R)$ and recover $v = m \parallel 0^{k_1}$ from $T = m \parallel 0^{k_1} \oplus G(R)$.
- If v ends in k_1 zeros output m, otherwise return $\perp$.

When applying the OAEP transform to produce an RSA-based version we take $k = 8 \cdot \lfloor \log_8(N) \rfloor$. We then produce the OAEP block A as above, and then we think of A as an integer less than N. It is to this integer we apply the RSA encryption function $f(A) = A^e \pmod N$. Upon decrypting we invert the function, by computing $f^{-1}(c) = c^d \pmod N$ to obtain an integer A'. We then check whether A' has the correct number of zero bits to the left, and if so take A as the k rightmost bits

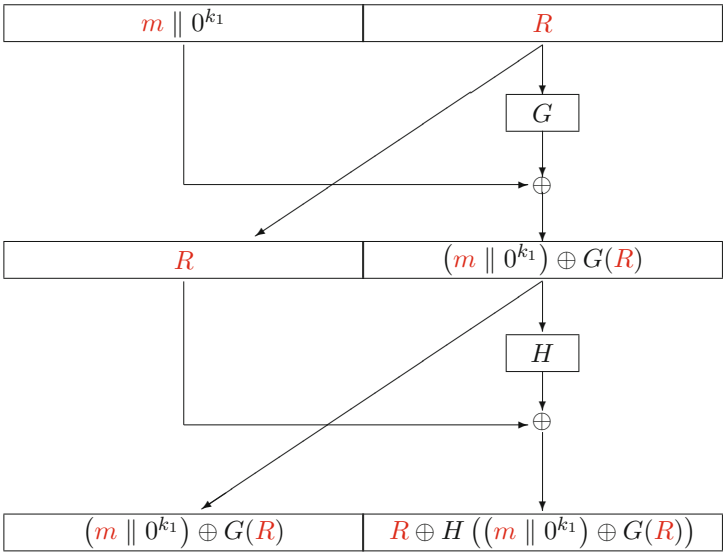


FIGURE 16.3. OAEP as a Feistel network

of A'. However, one needs to be careful about how one reports if the leftmost bits are not zero as expected. Such an error should not be distinguishable from the OAEP padding decoding returning a $\perp$. The main result about RSA-OAEP is the following.

Theorem 16.6. *In the random oracle model, if we model G and H by random oracles then RSA-OAEP is* IND-CCA *secure if the RSA problem is hard.*

PROOF. We sketch the proof and leave the details for the interested reader to look up. We first rewrite the RSA function f as

$$f : \begin{cases} \{0,1\}^{n+k_1} \times \{0,1\}^{k_0} & \longrightarrow & (\mathbb{Z}/N\mathbb{Z})^* \\ (s,t) & \longmapsto & (s\|t)^e \pmod N, \end{cases}$$

assuming the above-mentioned padding to the left is performed on encryption. We then define RSA-OAEP as applying the above function f to the inputs

$$s = (m\|0^{k_1}) \oplus G(r) \text{ and } t = r \oplus H(s).$$

The RSA assumption can be proved to be equivalent to the partial one-wayness of the function f, in the sense that the problem of recovering s from $f(s,t)$ is as hard as recovering (s,t) from $f(s,t)$. So for the rest of our sketch we try to turn an adversary A for breaking RSA-OAEP into an algorithm B which solves the partial one-wayness of the RSA function. In particular B is given $c^* = f(s^*,t^*)$, for some fixed RSA modulus N, and is asked to compute s^*.

Algorithm A works in the random oracle model and so it is assumed to only access the functions H and G via external calls, with the answers being provided by the environment. In addition A expects H and G to "act like" random functions. In our context B is the environment for A and so needs to supply A with the answers to its calls to the functions H and G. Thus B maintains a list of queries to H and G made by algorithm A, along with the responses. We call these lists the H-List and the G-List respectively.

Algorithm B now calls algorithm A, which will make a series of calls to the H and G oracles (we discuss how these are answered below). Eventually A will make a call to its $\mathcal{O}_{\mathsf{LR}}$ oracle by

producing two messages m_0 and m_1 of n bits in length. A bit b is then chosen by B, and B now assumes that c^* is the encryption of m_b. The ciphertext c^* is now returned to A, as the response to its call of the $\mathcal{O}_{\mathsf{LR}}$ oracle. Algorithm A then continues and tries to guess the bit b.

The oracle queries are answered by B as follows:

- Query $G(\gamma)$:
  For any query (δ, δ_H) in the H-List one checks whether
  $$c^* = f(\delta, \gamma \oplus \delta_H).$$
  – If this holds then we have partially inverted f as required (i.e. B can output δ as its solution). We can still, however, continue with the simulation of G and set
  $$G(\gamma) = \delta \oplus (m_b \| 0^{k_1}).$$
  – If this equality does not hold for any value of δ then we choose $\gamma_G = G(\gamma)$ uniformly at random from the codomain of G, and add the pair (γ, γ_G) to the G-List.
- Query $H(\delta)$:
  A random value δ_H is chosen from the codomain of H; the value (δ, δ_H) is added to the H-List. We also check whether for any (γ, γ_H) in the G-List we have
  $$c^* = f(\delta, \gamma \oplus \delta_H),$$
  if so we have managed to partially invert the function f as required, and we output the value of δ.
- Query decryption of c :
  We look in the G-List and the H-List for a pair $(\gamma, \gamma_G), (\delta, \delta_H)$ such that if we set $\sigma = \delta$, $\tau = \gamma \oplus \delta_H$ and $\mu = \gamma_G \oplus \delta$, then $c = f(\sigma, \tau)$ and the k_1 least significant bits of μ are equal to zero. If this is the case then we return the plaintext consisting of the n most significant bits of μ, otherwise we return $\perp$.

Notice that if a ciphertext which was generated in the correct way (by calling G, H and the encryption algorithm) is then passed to the above decryption oracle, we will obtain the original plaintext back.

We have to show that the above decryption oracle is able to "fool" the adversary A enough of the time. In other words when the oracle is passed a ciphertext which has not been generated by a prior call to the necessary G and H, we need to show that it produces a value which is consistent with the running of the adversary A. Finally we need to show that if the adversary A has a non-negligible chance of breaking the semantic security of RSA-OAEP then one has a non-negligible probability that B can partially invert f.

These last two facts are proved by careful analysis of the probabilities associated with a number of events. Recall that B assumes that $c^* = f(s^*, t^*)$ is an encryption of m_b. Hence, there should exist an r^* which satisfies

$$r^* = H(s^*) \oplus t^*,$$
$$G(r^*) = s^* \oplus (m_b \| 0^{k_1}).$$

One first shows that the probability of the decryption simulator failing is negligible. Then one shows that the probability that s^* is actually asked of the H oracle is non-negligible, as long as the adversary A has a non-negligible probability of finding the bit b. But as soon as s^* is asked of H then we spot this and can therefore break the partial one-wayness of f.

The actual technical probability arguments are rather involved and we refer the reader to the paper of Fujisaki, Okamoto, Pointcheval and Stern where the full proof is given. □

16.2.2. The Fujisaki–Okamoto Transform: We end this section with another generic transform which can be applied to one of the IND-CPA secure schemes from Section 16.1, to turn it into an IND-CCA secure encryption scheme. Like the OAEP transform, the transform in this section is secure assuming the adversary operates in the random oracle model.

Suppose we have a public key encryption scheme which is semantically secure against chosen plaintext attacks, such as ElGamal encryption. Such a scheme by definition needs to be non-deterministic hence we write the encryption function as

$$E(m, r),$$

where m is the message to be encrypted and r is the random input, and we denote the decryption function by $D(c)$. Hence, for ElGamal encryption we have

$$E(m, r) = (g^r, m \cdot h^r).$$

Fujisaki and Okamoto showed how to turn such a scheme into one which is IND-CCA secure. Their result only applies in the random oracle model and works by showing that the resulting scheme is plaintext aware. We do not go into the details of the proof at all, but simply give the transformation, which is both simple and elegant.

We take the encryption function above and alter it by setting

$$E'(m, r) = E(m\|r, H(m\|r))$$

where H is a hash function. The decryption algorithm is also altered in that we first compute

$$m' = D(c)$$

and then we check that

$$c = E(m', H(m')).$$

If this last equation holds we recover m from $m' = m\|r$; if the equation does not hold then we return $\perp$. For ElGamal encryption we therefore obtain the encryption algorithm

$$(g^{H(m\|r)}, (m\|r) \cdot h^{H(m\|r)}),$$

which is only marginally less efficient than raw ElGamal encryption.

16.3. Hybrid Ciphers

Almost always public key schemes are used only to transmit a short per message secret, such as a session key. This is because public key schemes are too inefficient to use to encrypt vast amounts of data. The actual data is then encrypted using a symmetric cipher. Such an approach is called a hybrid encryption scheme.

We now formalize this way of designing a public key encryption scheme with a hybrid cipher, via the so-called KEM/DEM approach. A KEM is a Key Encapsulation Mechanism, which is the public key component of a hybrid cipher, whilst a DEM is a Data Encapsulation Mechanism, which is the symmetric component. We have already mentioned DEMs in Chapters 11 and 13, where we constructed DEMs which were ot-IND-CCA secure as symmetric key encryption schemes, namely symmetric key schemes which were only ever designed to encrypt a single message.

We will present a security model for KEMs and then show, without using random oracles, that a suitably secure DEM and a suitably secure KEM can be combined to produce an IND-CCA secure hybrid cipher. This means we only need to consider the symmetric and public key parts separately, simplifying our design considerably. Finally, we show how a KEM can be constructed in the random oracle model using either the RSA or the DLP primitive. The resulting KEMs are very simple to construct and very natural, so we see the simplification obtained by utilizing hybrid encryption.

16.3.1. Defining a Key Encapsulation Mechanism: We define a Key Encapsulation Mechanism, or KEM, to be a mechanism which from the encryptor's side takes a public key $\mathfrak{pk}$ and outputs a symmetric key $k \in \mathbb{K}$ and an encapsulation of that key c for use by the holder of the corresponding private key. The holder of the private key $\mathfrak{sk}$ can then take the encapsulation c and their private key, and then recover the symmetric key k. Thus no message is input into the encapsulation mechanism. We therefore have three algorithms which operate as follows:

- $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \text{KeyGen}(\mathbb{K})$.
- $(c, k) \leftarrow \text{Encap}_{\mathfrak{pk}}()$.
- $k \leftarrow \text{Decap}_{\mathfrak{sk}}(c)$.

For correctness we require, for all pairs $(\mathfrak{pk}, \mathfrak{sk})$ output by $\text{KeyGen}(\mathbb{K})$, that

$$\text{If } (c, k) \leftarrow \text{Encap}_{\mathfrak{pk}}() \text{ then } \text{Decap}_{\mathfrak{sk}}(c) = k.$$

The security definition for KEMs is based on the security definition of indistinguishability of encryptions for public key encryption algorithms. However, we now require that the key output by a KEM should be indistinguishable from a random key. Thus the security game is defined via the following game.

- The challenger generates a random key $k_0 \in \mathbb{K}$ from the space of symmetric keys output by the KEM.
- The challenger calls the Encap function of the KEM to produce a valid key $k_1 \in \mathbb{K}$ and its encapsulation c^*, under the public key $\mathfrak{pk}$.
- The challenger picks a bit b and sends to the adversary the values k_b, c^*.
- The goal of the adversary is to decide whether $b = 0$ or 1.

The advantage of the adversary, against the KEM Π, is defined to be

$$\text{Adv}_{\Pi}^{\text{IND-CPA}}(A) = 2 \cdot \left| \Pr(A(\mathfrak{pk}, k_b, c^*) = b) - \frac{1}{2} \right|.$$

The above only defines the security in the passive case; to define security under adaptive chosen ciphertext attacks one needs to give the adversary access to a decapsulation function. This decapsulation function will return the key (or the invalid encapsulation symbol) for any encapsulation of the adversary's choosing, bar the target encapsulation c^*. In such a situation we denote the advantage by $\text{Adv}_{\Pi}^{\text{IND-CCA}}(A)$, and say the KEM is secure if this advantage is "small" for all adversaries A. We describe the full security model in Figure 16.4.

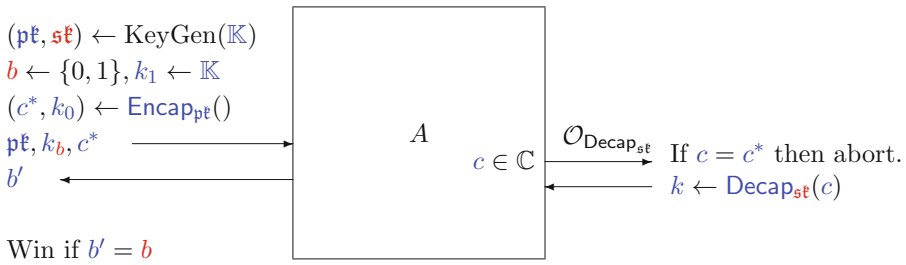


FIGURE 16.4. Security game IND-CCA for a KEM

16.3.2. Generically Constructing Hybrid Encryption: The idea of a KEM/DEM system is that one takes a KEM (defined by the algorithms $\text{KeyGen}, \text{Encap}_{\mathfrak{pk}}, \text{Decap}_{\mathfrak{sk}}$ and which outputs symmetric keys from the space $\mathbb{K}$) and a DEM (defined by the algorithms e_k, d_k and with key space $\mathbb{K}$), and then uses the two together to form a hybrid cipher, a.k.a. a public key encryption

scheme. The key generation method of the public key scheme is simply the key generation method of the underlying KEM. To encrypt a message m to a user with public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$, one performs the following steps:

- $(k, c_1) \leftarrow \mathsf{Encap}_{\mathfrak{pk}}()$.
- $c_2 \leftarrow e_k(m)$.
- $c \leftarrow (c_1, c_2)$.

The recipient, upon receiving the pair $c = (c_1, c_2)$, performs the following steps to recover m.

- $k \leftarrow \mathsf{Decap}_{\mathfrak{sk}}(c_1)$.
- If $k = \perp$ return $\perp$.
- $m \leftarrow d_k(c_2)$.
- Return m.

We would like the above hybrid cipher to meet our security definition for public key encryption schemes, namely IND-CCA.

Theorem 16.7. *The hybrid public key encryption scheme Π defined above is* IND-CCA *secure, assuming the KEM scheme Π_1 is* IND-CCA *secure and the DEM Π_2 is* ot-IND-CCA *secure. In particular if A is an adversary against the* IND-CCA *security of the hybrid public key encryption scheme then there exist adversaries B and C such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) \leq 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C).$$

Before we give the proof notice that we only need one-time security for the DEM, as each symmetric encryption key output by the KEM is only used once. Thus we can construct the DEM from much simpler components.

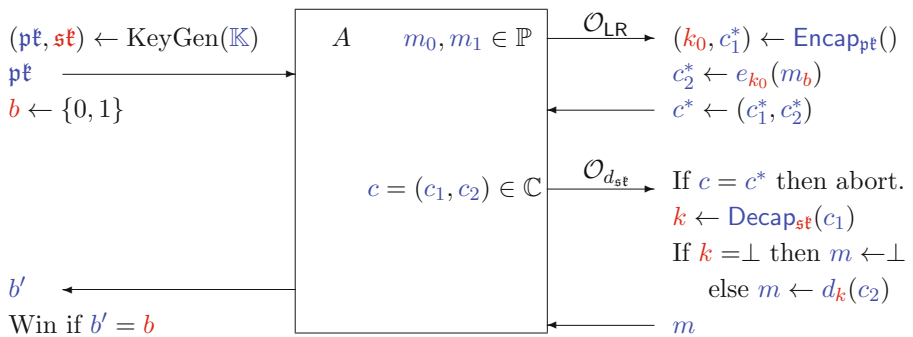


FIGURE 16.5. IND-CCA game G_0 for our hybrid scheme

PROOF. We sketch the proof and direct the reader to the paper of Cramer and Shoup for more details. First consider Figure 16.5; this is the standard IND-CCA game for public key encryption, tailored for our hybrid encryption scheme. Let us call this game G_0. We now modify the game which A is playing; instead of encrypting c_2^* using the valid key k_0 we instead use a new random key called k_1. This modified game we call G_1 and we present it in Figure 16.6.

Game G_1 is relatively easy to analyse so we will do this first. We will show that if A wins in game G_1, then we can construct an adversary C which will use A as a subroutine to break the ot-IND-CCA security of the DEM (e_k, d_k). The key trick we use is that the c_1^* component in game G_1 is unrelated to the key which is used to encrypt m_b. We present algorithm C in Algorithm 16.3. Notice that in Game G_1 when A makes a decryption query for (c_1, c_2), it is validly decrypted by C, as long as $c_1 \neq c_1^*$. When this last condition holds, algorithm C uses its own decryption

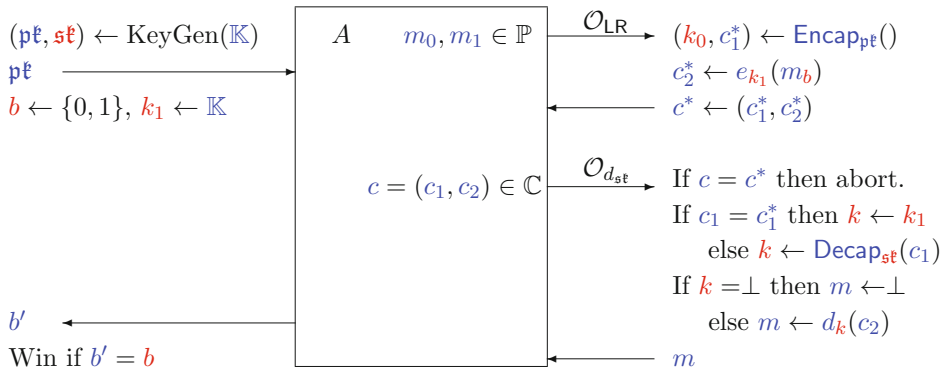


FIGURE 16.6. Game G_1

oracle to return the decryption of c_2. Note that B's target ciphertext c_2^* is never passed to its own decryption oracle, unless B aborts because A made an invalid query. In addition, note that the $\mathcal{O}_{\mathsf{LR}}$ oracle of C is only called once, as is required in the one-time security of a DEM. Finally, note that A winning (or losing) game G_1 directly corresponds to C winning (or losing) game, thus

$$\mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C) = 2 \cdot \left| \Pr[C \text{ wins }] - \frac{1}{2} \right| = 2 \cdot \left| \Pr[A \text{ wins in game } G_1] - \frac{1}{2} \right|.$$

Algorithm 16.3: Algorithm C

$(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathrm{KeyGen}(\mathbb{K})$.
Call A with input the public key $\mathfrak{pk}$.

/* A's $\mathcal{O}_{\mathsf{LR}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{LR}}$ query with messages m_0, m_1.
C passes m_0, m_1 to its own $\mathcal{O}_{\mathsf{LR}}$ oracle to obtain c_2^*.
$c_1^* \leftarrow \mathsf{Encap}_{\mathfrak{pk}}()$.
$c^* \leftarrow (c_1^*, c_2^*)$.
return c^*.

/* A's $\mathcal{O}_{d_k}$ Oracle Queries */
A makes an $\mathcal{O}_{d_k}$ query with ciphertext $c = (c_0, c_1)$.
if $c_0 \neq {c_0}^*$ **then** $k \leftarrow \mathsf{Decap}_{\mathfrak{sk}}(c_0)$, $m \leftarrow d_k(c_1)$.
else if $c_1 \neq {c_1}^*$ **then** C passes c_1 to its $\mathcal{O}_{d_k}$ oracle to obtain m.
else abort.
return m.

/* A's Response */
When A returns b'.
return b'.

We now turn to what is the most complex step. We want to bound the probability

$$\left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right|.$$

We do this by presenting an algorithm B which uses A to break the IND-CCA security of the KEM. It does this as follows: Algorithm B does not know whether the key/encapsulation pair given to it is a real encapsulation of the key, or a fake one. It constructs an environment for A to play in, where in the first case A is playing game G_0, whereas in the second it is playing game G_1. Hence, any advantage A has in distinguishing the two games it is playing in, can be exploited by B to break the KEM. We give the algorithm for B in Algorithm 16.4.

Algorithm 16.4: Algorithm B

B has as input $\mathfrak{pk}$.
Call A with input the public key $\mathfrak{pk}$.

/* A's $\mathcal{O}_{\mathsf{LR}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{LR}}$ query with messages (m_0, m_1).
B calls its own $\mathcal{O}_{\mathsf{LR}}$ oracle to obtain (c_1^*, k^*).
$b \leftarrow \{0,1\}$.
$c_2^* \leftarrow e_{k^*}(m_b)$.
$c^* \leftarrow (c_1^*, c_2^*)$.
return c^*

/* A's $\mathcal{O}_{d_k}$ Oracle Queries */
A makes an $\mathcal{O}_{d_k}$ query with ciphertext $c = (c_0, c_1)$.
if $c_0 \neq {c_0}^*$ **then**
  B calls its $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ oracle on c_0 to obtain k.
  $m \leftarrow d_k(c_1)$.
else if $c_1 \neq {c_1}^*$ **then** $m \leftarrow d_{k^*}(c_1)$.
else abort.
return m.

/* A's Response */
When A returns b'.
$a \leftarrow 0$.
if $b' \neq b$ **then** $a \leftarrow 1$.
return a.

So algorithm B outputs zero if it thinks A is playing in game G_0, i.e. if k^* is the actual key underlying the encapsulation c_1^*, whereas B will output one if it thinks A is playing in game G_1, i.e. if k^* is just some random key. So we have

$$\begin{aligned} 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) &= 2 \cdot \Big| \Pr[a = 0 \mid A \text{ in game } G_0] - \Pr[a = 0 \mid A \text{ in game } G_1] \Big| \\ &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \Big|. \end{aligned}$$

We then have that

$$\begin{aligned} \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \frac{1}{2} \Big| \\ &= 2 \cdot \Big| \Pr[A \text{ wins in game } G_0] - \frac{1}{2} \\ &\qquad - \Pr[A \text{ wins in game } G_1] + \Pr[A \text{ wins in game } G_1] \Big| \qquad \text{adding zero} \end{aligned}$$

$$
\begin{aligned}
&\leq 2 \cdot \left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right| \\
&\quad + 2 \cdot \left| \Pr[A \text{ wins in game } G_1] - \frac{1}{2} \right| && \text{triangle Inequality} \\
&= 2 \cdot \left| \Pr[A \text{ wins in game } G_0] - \Pr[A \text{ wins in game } G_1] \right| \\
&\quad + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C) \\
&= 2 \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi_1}(B) + \mathrm{Adv}^{\mathsf{ot\text{-}IND\text{-}CCA}}_{\Pi_2}(C).
\end{aligned}
$$

□

16.4. Constructing KEMs

As previously mentioned, KEMs are simpler to design than public key encryption algorithms. In this section we first look at RSA-KEM, whose construction and proof should be compared to that of RSA-OAEP. Then we turn to DHIES-KEM which should be compared to the ElGamal variant of the scheme based on the Fujisaki–Okamoto transform.

16.4.1. RSA-KEM: Let N denote an RSA modulus, i.e. a product of two primes p and q of roughly the same size. Let e denote an RSA public exponent and d an RSA private exponent. We let $f_{N,e}(x)$ denote the RSA function, i.e. the function that maps an integer x modulo N to the number $x^e \pmod N$. The important point for RSA-KEM is that this is a trapdoor one-way function; only the holder of the secret trapdoor d should be able to invert it. This is summarized in the RSA problem, which is the problem of given an integer y modulo N to recover the value of x such that $f_{N,e}(x) = y$.

We define RSA-KEM by taking a cryptographic key derivation function H which takes integers modulo N and maps them to symmetric keys of the size required by the user of the KEM (i.e. the key size of the DEM). In our security model we will assume that H behaves like a random oracle. Encapsulation then works as follows:

- $x \leftarrow \{1, \ldots, N-1\}$.
- $c \leftarrow f_{N,e}(x)$.
- $k \leftarrow H(x)$.
- Output (k, c).

Since the person with the private key can invert the function $f_{N,e}$, decapsulation is easily performed via

- $x \leftarrow f_N^{-1}(c)$.
- $k \leftarrow H(x)$.
- Output k.

There is no notion of invalid ciphertexts, and this is simpler in comparison to RSA-OAEP. The construction actually works for *any* trapdoor one-way function. We now only need to show that this simple construction meets our definition of a secure KEM.

Theorem 16.8. *In the random oracle model RSA-KEM is an* IND-CCA *secure KEM, assuming the RSA problem is hard. In particular given an adversary A against the* IND-CCA *property of the RSA-KEM scheme Π, for moduli of size v bits, which treats H as a random oracle, then there is an adversary B against the RSA problem for integers of size v such that*

$$\mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\Pi}(A) \leq \mathrm{Adv}^{\mathsf{RSA}}_{v}(B).$$

Proof. Since A works in the random oracle model, we model the function H in the proof as a random oracle. Thus algorithm B keeps a list of triples (z, c, h), which we will call the H-List, of queries to H, which is initially set to be empty. The value z denotes the query to the function H,

the value h the output and the value c denotes the output of $f_{N,e}$ on z. Algorithm B has as input a value y for which it is trying to invert the function $f_{N,e}$. Algorithm B passes the values N, e to A as the public key of the KEM which A is trying to attack. To generate the challenge encapsulation, the challenger generates a symmetric key k at random and takes as the challenge encapsulation the value $c^* \leftarrow y$ of the RSA function for which B is trying to find the preimage. It then passes k and c^* to A.

The adversary is allowed to make queries of H for values z. If this query on z has been made before, then B uses its H-List to respond as required. If there is a value on the list of the form $(\perp, c, h)$ with $f_{N,e}(z) = c$ then B replaces this value with (z, c, h) and responds with h. Otherwise B generates a new random value of h, adds the triple $(z, f_{N,e}(z), h)$ to the list and responds with h.

The adversary can also make decapsulation queries on an encapsulation c. If there is a value (z, c, h), for some c and h, on the H-List it responds with h. Otherwise, it generates h at random, places the triple $(\perp, c, h)$ on the list and responds with h.

Since A is running in the random oracle model, the only way that A can have any success in the game is by querying H on the preimage of y. Thus if A is successful then the preimage of y will exist on the list of triples kept by algorithm B. Hence, when A terminates B searches its H-List for a triple of the form (x, y, h) and if there is one it outputs x as the preimage of y.

In summary algorithm B is presented in Algorithm 16.5. It is easy to see that the calls to H and the calls to $\mathcal{O}_{\mathsf{Decap}_{s\mathfrak{k}}}$ are answered by B in a consistent way, due to algorithm B's ability to ensure the required behaviour of the random oracle responses. □

16.4.2. The DHIES Encryption Scheme: The DHIES encryption scheme is the instantiation of our hybrid encryption paradigm, with the DHIES-KEM and the data encapsulation mechanism being the Encrypt-then-MAC instantiation. The scheme was designed by Abdalla, Bellare and Rogaway and was originally called DHAES, for Diffie–Hellman Augmented Encryption Scheme. However, this caused confusion with the Advanced Encryption Standard. So the name was changed to DHIES, for Diffie–Hellman Integrated Encryption Scheme. When used with elliptic curves it is called ECIES. To define the scheme all we need to do is present the DHIES-KEM component, as the rest follows from our prior discussions.

Key Generation: The domain parameters are a cyclic finite abelian group G of prime order q, a generator g and the key space $\mathbb{K}$ for the data encapsulation mechanism to be used. We require a key derivation function H with codomain equal to $\mathbb{K}$, which again we will model as a random oracle. To generate a public/private key pair we generate a random $x \leftarrow \mathbb{Z}/q\mathbb{Z}$ and compute the public key $h \leftarrow g^x$.

Encapsulation: Encapsulation proceeds as follows:

- $u \leftarrow \mathbb{Z}/q\mathbb{Z}$.
- $v \leftarrow h^u$.
- $c \leftarrow g^u$.
- $k \leftarrow H(v \| c)$.

Decapsulation: To decapsulate the KEM one takes c and using the private key one computes

- $v \leftarrow c^x$.
- $k \leftarrow H(v \| c)$.

However, to prove this KEM secure we need to introduce a new problem called the Gap Diffie–Hellman problem. This problem assumes that the Diffie–Hellman problem is hard even assuming that the adversary has an oracle to solve the Decision Diffie–Hellman problem. In other words, we are given g^a and g^b and an oracle $\mathcal{O}_{\mathsf{DDH}}$ which on input of (g^x, g^y, g^z) will say whether $z = x \cdot y$. We

Algorithm 16.5: Algorithm B using an $\mathsf{IND\text{-}CCA}$ adversary A against RSA-KEM to solve the RSA problem

B has input N, e, y and is asked to find x such that $x^e \pmod N = y$.
$(\mathfrak{pk}) \leftarrow (N, e)$.
$k \leftarrow \mathbb{K}$, $c^* \leftarrow y$.
H-List $\leftarrow \emptyset$.
Call A with input $\mathfrak{pk}, k, c^*$.

/* A's H Oracle Queries */
A makes a random oracle query with input z.
if $\exists\ (z, c, h) \in H$-List **then return** h.
if $\exists\ (\perp, c, h) \in H$-List with $z^e \pmod N = c$ **then**
  $H\text{-List} \leftarrow \Big(H\text{-List} \cup \{(z, c, h)\}\Big) \setminus \{(\perp, c, h)\}$.
else
  $h \leftarrow \mathbb{K}$.
  $c \leftarrow z^e \pmod N$.
  $H\text{-List} \leftarrow H\text{-List} \cup \{(z, c, h)\}$.
return h

/* A's $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ query with ciphertext c.
if $\exists\ (\cdot, c, h) \in H$-List **then return** h
$h \leftarrow \mathbb{K}$.
$H\text{-List} \leftarrow H\text{-List} \cup \{(\perp, c, h)\}$.
return h.

/* A's Response */
When A returns b'.
if $\exists\ (x, c^*, \cdot) \in H$-List **then return** x.
return $\perp$.

then wish to output $g^{a \cdot b}$. We define $\mathrm{Adv}_G^{\mathsf{Gap\text{-}DHP}}(A)$ as the probability that the algorithm A wins the Diffie–Hellman problem game, given access to an oracle which solves the Decision Diffie–Hellman problem. It is believed that this problem is as hard as the standard Diffie–Hellman problem. Indeed there are some groups in which the Decision Diffie–Hellman problem is easy and the computational Diffie–Hellman problem is believed to be hard.

We can now prove that the DHIES-KEM is secure. Before stating and proving the theorem we pause to point out why we need the Gap Diffie–Hellman problem. In the proof of security of RSA-KEM, Theorem 16.8, the algorithm B's simulation of a valid attack environment to algorithm A was perfect. In other words A could not notice it was playing against someone trying to solve the RSA problem, and not a genuine encryption system. Algorithm B did this by "cooking" the values output by the random oracle, by computing the RSA function in a forwards direction. In the simulation in the theorem below, algorithm B still proceeds with much the same strategy. However, to do a similar cooking of the random oracle algorithm B needs to be able to distinguish Diffie–Hellman tuples, from non-Diffie–Hellman tuples. Thus algorithm B needs a mechanism to do this. Hence, algorithm B needs access to a $\mathcal{O}_{\mathsf{DDH}}$ oracle, and so B does not solve the Diffie–Hellman problem, but the Gap Diffie–Hellman problem.

Theorem 16.9. *In the random oracle model and assuming the Gap Diffie–Hellman problem is hard, there exists no adversary which breaks DHIES-KEM. In particular if A is an adversary which breaks the* IND-CCA *security of the DHIES-KEM scheme Π for the group G, which treats H as a random oracle, then there is an adversary B against the Gap Diffie–Hellman problem for the group G with*

$$\mathrm{Adv}_{\Pi}^{\mathsf{IND\text{-}CCA}}(A) = \mathrm{Adv}_{G}^{\mathsf{Gap\text{-}DHP}}(B).$$

Proof. We provide a sketch of the proof by simply giving algorithm B in Algorithm 16.6, and presenting some comments. Notice that the public key is g^a and the target encapsulation is $c^* = g^b$. Hence, the Diffie–Hellman value which needs to be passed to the key derivation function H to obtain the target encapsulated key is $v^* = g^{a\cdot b}$. Since H is a random oracle the only way A can find out any information about the encapsulated key is to make the query $H(v^*\|c^*)$. Thus the Diffie–Hellman value, if A is successful, will end up on B's H-List. When looking at Algorithm 16.6 you should compare it with Algorithm 16.5.

Algorithm 16.6: Algorithm B using an IND-CCA adversary A against DHIES-KEM to solve the Gap Diffie–Hellman problem

B has input $A = g^a$, $B = g^b$ and is asked to find $C = g^{a\cdot b}$.
$(\mathfrak{pk}) = h \leftarrow A$.
$k \leftarrow \mathbb{K}$, $c^* \leftarrow B$.
H-List $\leftarrow \emptyset$.
Call A with input $\mathfrak{pk}, k, c^*$.

/* A's H Oracle Queries */
A makes a random oracle query with input $z\|c$.
if $\exists\ (z, c, h) \in H$-List **then return** h.
if $\exists(\perp, c, h) \in H$-List such that $\mathcal{O}_{\mathsf{DDH}}(g, A, c, z)$=**true then**
  H-List $\leftarrow \Big(H\text{-List} \cup \{(z, c, h)\}\Big) \setminus \{(\perp, c, h)\}$.
else
  $h \leftarrow \mathbb{K}$.
  H-List $\leftarrow H$-List $\cup \{(z, c, h)\}$.
return h.

/* A's $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ Oracle Queries */
A makes an $\mathcal{O}_{\mathsf{Decap}_{\mathfrak{sk}}}$ query with ciphertext c.
if $\exists\ (\cdot, c, h) \in H$-List **then return** h.
$h \leftarrow \mathbb{K}$.
H-List $\leftarrow H$-List $\cup \{(\perp, c, h)\}$.
return h.

/* A's Response */
When A returns b'.
if $\exists\ (C, c^*, \cdot) \in H$-List **then return** C
return $\perp$.

□

16.5. Secure Digital Signatures

We already saw in Chapter 15 how the combination of hash functions and the RSA function could be used to produce digital signatures. In particular we presented the RSA-FDH signature scheme. In this section we first prove that this scheme is secure in the random oracle model. However, RSA-FDH requires a hash function with codomain the RSA group. Since such hash functions are not "natural" we also present the RSA-PSS signature scheme which does not have this restriction, and which is also secure in the random oracle model. Having presented these two variants of signatures based on the RSA problem, we then turn to discussing signature schemes based on the discrete logarithm problem.

16.5.1. RSA-FDH: In Chapter 15 we presented the RSA-FDH signature scheme. The proof we outline below for RSA-FDH bears much in common with the proof for RSA-KEM above, especially in the way the hash function is modelled as a random oracle. For RSA-FDH we assume a hash function

$$H : \{0,1\}^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^*,$$

where N is the RSA modulus of the public key. Again such hash functions are hard to construct in practice, but if we assume they can exist and we model them using a random oracle then we can prove the RSA-FDH signature algorithm is secure.

As above let $f_{N,e}$ denote the function

$$f_{N,e} : \begin{cases} (\mathbb{Z}/N\mathbb{Z})^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^* \\ x \longmapsto x^e. \end{cases}$$

The RSA-FDH signature algorithm signs a message m as follows

$$s \leftarrow H(m)^d \pmod N = f_{N,e}^{-1}(H(m)),$$

where the private exponent is d. Verification of a signature is performed by checking whether

$$f_{N,e}(s) = s^e \pmod N = H(m).$$

Recall that the RSA problem is given $y = f_{N,e}(x)$ determine x. One can then prove the following theorem.

Theorem 16.10. *In the random oracle model if we model H as a random oracle then the RSA-FDH signature scheme is secure, assuming the RSA problem is hard. In particular if A is* EUF-CMA *adversary against the RSA-FDH signature scheme Π for RSA moduli of v bits in length which performs q_H distinct hash function queries, then there is an algorithm B for the RSA problem such that*

$$\mathrm{Adv}_{\Pi}^{\mathsf{EUF\text{-}CMA}}(A; q_H) = q_H \cdot \mathrm{Adv}_{v}^{\mathsf{RSA}}(B).$$

Note that in this theorem the advantage term has a "security loss" of q_H, this is because algorithm B in the proof needs to "guess" into which query it should embed the RSA problem challenge.

Proof. We describe an algorithm B which on input of $y \in (\mathbb{Z}/N\mathbb{Z})^*$ outputs $x = f_{N,e}^{-1}(y)$. Without loss of generality we can assume that algorithm A always makes a hash function query on a message m before it is passed to its signing oracle. Indeed, if this is not the case then B can make these queries for A.

Algorithm B first chooses a value $t \in [1, \ldots, q_H]$ and throughout keeps a numbered record of all the hash queries made. Algorithm B takes as input N, e and y and sets the public key to be $pk \leftarrow (N, e)$. Algorithm B maintains a hash list H-List as before, which is initially set to the empty set. The public key is then passed to algorithm A.

When algorithm A makes a hash function query for the input m, algorithm B responds as follows:

- If there exists $(m, s, \perp) \in H$-List then return s.

- If this is the tth distinct query to the hash function then B sets
$$H\text{-List} \leftarrow H\text{-List} \cup \{(m, y, \perp)\}$$
and responds with y. We let m^* denote this message.
- Else B picks $s \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$, sets $h \leftarrow s^e \pmod N$ and sets
$$H\text{-List} \leftarrow H\text{-List} \cup \{(m, h, s)\}$$
and responds with h.

If A makes a signing query for a message m then algorithm B responds as follows.

- If message m is equal to m^* then algorithm B stops and returns fail.
- If $m \neq m^*$ then B returns the value s such that $(m, h, s) \in H\text{-List}$.

Let A terminate with output (m, s) and without loss of generality we can assume that A made a hash oracle query for the message m. Now if $m \neq m^*$ then B terminates and admits failure, but if $m = m^*$ then we have
$$f_{N,e}(s) = H(m_t) = y.$$
Hence we have succeeded in inverting f.

In analysing algorithm B one notices that if A terminates successfully then (m^*, s) is an existential forgery and so m^* was not asked of the signing oracle. The value of t is independent of the view of A, so A cannot always ask for the signature of message m^* in the algorithm rather than not ask for the signature. Hence, roughly speaking, the probability of success of B is $1/q_H$ that of the probability of A being successful. □

16.5.2. RSA-PSS: Another way of securely using RSA as a signature algorithm is to use a system called RSA-PSS, or *probabilistic signature scheme*. This scheme can also be proved secure in the random oracle model under the assumption that the RSA problem is hard. We do not give the details of the proof here but simply explain the scheme, which appears in many cryptographic standards. The advantage of RSA-PSS over RSA-FDH is that one only requires a hash function with a traditional codomain, e.g. bit strings of length t, rather than a set of integers modulo another number.

As usual one takes an RSA modulus N, a public exponent e and a private exponent d. Suppose the security parameter is k, i.e. N is a k-bit number. We define two integers k_0 and k_1 so that $k_0 + k_1 \leq k - 1$, such that a work effort of 2^{k_0} and 2^{k_1} is considered infeasible; for example one could take $k_i = 128$ or 160. We then define two hash functions, one which expands data and one which compresses data (just like in RSA-OAEP):
$$G : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_1-1}$$
$$H : \{0,1\}^* \longrightarrow \{0,1\}^{k_1}.$$
We let
$$G_1 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k_0}$$
denote the function which returns the first k_0 bits of $G(w)$ for $w \in \{0,1\}^{k_1}$ and we let
$$G_2 : \{0,1\}^{k_1} \longrightarrow \{0,1\}^{k-k_0-k_1-1}$$
denote the function which returns the last $k - k_0 - k_1 - 1$ bits of $G(w)$ for $w \in \{0,1\}^{k_1}$, i.e. $G(w) = G_1(w) \| G_2(w)$.

Signing: To sign a message m the private key holder performs the following steps:

- $r \leftarrow \{0,1\}^{k_0}$.
- $w \leftarrow H(m \| r)$.
- $y \leftarrow 0 \| w \| (G_1(w) \oplus r) \| G_2(w)$.
- $s \leftarrow y^d \pmod N$.

Verification: To verify a signature (s, m) the public key holder performs the following

- $y \leftarrow s^e \pmod N$.
- Split y into the components
$$b\|w\|\alpha\|\gamma$$
where b is one bit long, w is k_1 bits long, α is k_0 bits long and γ is $k - k_0 - k_1 - 1$ bits long.
- $r \leftarrow \alpha \oplus G_1(w)$.
- The signature is verified as correct if and only if $b = 0$ and $G_2(w) = \gamma$ and $H(m\|r) = w$.

If we allow the modelling of the hash functions G and H by random oracles then one can show that the above signature algorithm is EUF-CMA secure, in the sense that the existence of a successful algorithm to find forgeries could be used to produce an algorithm to invert the RSA function. For the proof of this one should consult the Eurocrypt 1996 paper of Bellare and Rogaway mentioned in the Further Reading section at the end of this chapter.

16.5.3. The Digital Signature Algorithm: We have already presented two secure digital signature schemes, namely RSA-FDH and RSA-PSS. You may ask why do we need another one?

- What if someone breaks the RSA algorithm or finds that factoring is easy?
- RSA is not suited to some applications since signature generation is a very costly operation.
- RSA signatures are very large; some applications require smaller signature footprints.

One algorithm which addresses all of these concerns is the Digital Signature Algorithm, or DSA. One sometimes sees this referred to as the DSS, or Digital Signature Standard. Although originally designed to work in the group $\mathbb{F}_p^*$, where p is a large prime, it is now common to see it used with elliptic curves, in which case it is called EC-DSA. The elliptic curve variants of DSA run very fast and have smaller footprints and key sizes than almost all other signature algorithms. We shall first describe the basic DSA algorithm as it applies to finite fields. In this variant the security is based on the difficulty of solving the discrete logarithm problem in the field $\mathbb{F}_p$.

Domain Parameters: Just as in ElGamal encryption we first need to define the domain parameters, which are identical to those used in ElGamal encryption. These are

- p a 'large prime', by which we mean one with around 2048 bits, such that $p-1$ is divisible by another 'medium prime' q of around 256 bits.
- g an element of $\mathbb{F}_p^*$ of prime order q, i.e. $g = r^{(p-1)/q} \pmod p \neq 1$ for some $r \in \mathbb{F}_p^*$.
- A hash function H which maps bit strings to element in $\mathbb{Z}/q\mathbb{Z}$.

The domain parameters create a public finite abelian group G of prime order q with generator g. Such domain parameters can be shared between a large number of users.

Key Generation: Again key generation is exactly the same as in ElGamal encryption. The private key $\mathfrak{sk}$ is chosen to be an integer $x \leftarrow [0, \ldots, q-1]$, whilst the public key is given by $\mathfrak{pk} = h \leftarrow g^x \pmod p$.

Signing: DSA is a signature with appendix algorithm and the signature produced consists of two elements $r, s \in \mathbb{Z}/q\mathbb{Z}$. To sign a message m the user performs the following steps:

- $h \leftarrow H(m)$.
- $k \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $r \leftarrow (g^k \pmod p) \pmod q$.
- $s \leftarrow (h + x \cdot r)/k \pmod q$.

The signature on m is then the pair (r, s). Notice that to sign we utilize a secret ephemeral key on every signature. One issue with DSA is that this ephemeral key k really needs to be kept secret and truly random, otherwise attacks like the earlier partial key exposure attacks from Section 15.5 can be deployed.

Verification: To verify the signature (r, s) on the message m for the public key h, the verifier performs the following steps.

- $h \leftarrow H(m)$.
- $a \leftarrow h/s \pmod q$.
- $b \leftarrow r/s \pmod q$.
- $v \leftarrow (g^a \cdot h^b \pmod p) \pmod q$.
- Accept the signature if and only if $v = r$.

As a baby example of DSA consider the following domain parameters

$$q = 13,\ p = 4 \cdot q + 1 = 53 \text{ and } g = 16.$$

Suppose the public/private key pair of the user is given by $x \leftarrow 3$ and $h \leftarrow g^3 \pmod p = 15$. Now, if we wish to sign a message which has hash value $h = 5$, we first generate an ephemeral secret key, for example purposes we shall take $k \leftarrow 2$, and then we compute

$$\begin{aligned} r &\leftarrow (g^k \pmod p) \pmod q = 5, \\ s &\leftarrow (h + x \cdot r)/k \pmod q = 10. \end{aligned}$$

To verify this signature the recipient computes

$$\begin{aligned} a &\leftarrow h/s \pmod q = 7, \\ b &\leftarrow r/s \pmod q = 7, \\ v &\leftarrow (g^a \cdot y^b \pmod p) \pmod q = 5. \end{aligned}$$

Note $v = r$ and so the signature is verified correctly.

The DSA algorithm uses the subgroup of $\mathbb{F}_p^*$ of order q which is generated by g. The private key can clearly be recovered from the public key if the discrete logarithm problem can be solved in the cyclic group $\langle g \rangle$ of order q. Thus, taking into account our discussion of the discrete logarithm problem in Chapter 3, we require for security that

- $p > 2^{2048}$, to avoid attacks via the Number Field Sieve,
- $q > 2^{256}$ to avoid attacks via the Baby-Step/Giant-Step method.

Hence, to achieve the rough equivalent of 128 bits of AES strength we need to operate on integers of roughly 2048 bits in length. This makes DSA slower than RSA, since the DSA operation is more complicated than RSA. For example, the verification operation for an equivalent RSA signatures requires only one exponentiation modulo a 2048-bit number, and even that is an exponentiation by a small number. For DSA, verification requires two exponentiations modulo a 2048-bit number. In addition the signing operation for DSA is more complicated than the procedure for RSA signatures, due to the need to compute the value of s, which requires an inversion modulo q.

The other main problem is that the DSA algorithm really only requires to work in a finite abelian group of size 2^{256}, but since the integers modulo p is susceptible to an attack from the Number Field Sieve we are required to work with group elements of 2048 bits in size. This produces a significant performance penalty.

Luckily we can generalize DSA to an arbitrary finite abelian group in which the discrete logarithm problem is hard. We can then use a group which provides a harder instance of the discrete logarithm problem, for example the group of points on an elliptic curve over a finite field. To explain this generalization, we write $G = \langle g \rangle$ for a group generated by g; we assume that

- g has prime order $q > 2^{256}$,
- the discrete logarithm problem with respect to g is hard,

- there is a public function f such that

$$f : G \longrightarrow \mathbb{Z}/q\mathbb{Z}.$$

We summarize the differences between DSA and EC-DSA in the following table.

Quantity	DSA	EC-DSA
G	$\langle g \rangle < \mathbb{F}_p^*$	$\langle P \rangle < E(\mathbb{F}_p)$
g	$g \in \mathbb{F}_p^*$	$P \in E(\mathbb{F}_p)$
y	g^x	$[x]P$
$f(\cdot)$	$\cdot \pmod q$	x-coord$(\cdot) \pmod q$

For this generalized form of DSA each user again generates a secret signing key, x. The public key is again given by $h \leftarrow g^x$. Signatures are computed via the steps

- $h \leftarrow H(m)$.
- $k \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $r \leftarrow f(g^k)$.
- $s \leftarrow (h + x \cdot r)/k \pmod q$.

To verify the signature (r, s) on the message m the verifier performs the following steps.

- $h \leftarrow H(m)$.
- $a \leftarrow h/s \pmod q$.
- $b \leftarrow r/s \pmod q$.
- $v \leftarrow f(g^a \cdot h^b)$.
- Accept the signature if and only if $v = r$.

You should compare this signature and verification algorithm with that given earlier for DSA and spot where they differ. When used for EC-DSA the above generalization is written additively.

EC-DSA Example: As a baby example of EC-DSA take the following elliptic curve

$$Y^2 = X^3 + X + 3,$$

over the field $\mathbb{F}_{199}$. The number of elements in $E(\mathbb{F}_{199})$ is equal to $q = 197$ which is a prime; the elliptic curve group is therefore cyclic and as a generator we can take $P = (1, 76)$. As a private key let us take $x = 29$, and so the associated public key is given by

$$Y = [x]P = [29](1, 76) = (113, 191).$$

Suppose the holder of this public key wishes to sign a message with hash value $H(m)$ equal to 68. They first produce a random ephemeral key, which we shall take to be $k = 153$, and compute

$$\begin{aligned} r &= x\text{-coord}\left([k]P\right) = x\text{-coord}\left([153](1, 76)\right) \\ &= x\text{-coord}\left((185, 35)\right) = 185. \end{aligned}$$

Now they compute

$$\begin{aligned} s &= (H(m) + x \cdot r)/k \pmod q \\ &= (68 + 29 \cdot 185)/153 \pmod{197} \\ &= 78. \end{aligned}$$

The signature is then the pair $(r, s) = (185, 78)$.

To verify this signature we compute

$$\begin{aligned} a &= H(m)/s \pmod q = 68/78 \pmod{197} = 112, \\ b &= r/s \pmod q = 185/78 \pmod{197} = 15. \end{aligned}$$

We then compute

$$\begin{aligned} Z &= [a]P + [b]Y = [112](1,76) + [15](113,191) \\ &= (111,60) + (122,140) = (185,35). \end{aligned}$$

The signature is now verified since we have

$$r = 185 = x\text{-coord}(Z).$$

It is believed that DSA and EC-DSA do provide secure signature algorithms, in the sense of EUF-CMA, however no proof of this fact is known in the standard model or the random oracle model. However, if instead of modelling the hash function as an ideal object (as in the random oracle model), we model the group as an ideal object (something called the generic group model) then we can show that EC-DSA is EUF-CMA secure. But this uses techniques beyond the scope of this book.

16.5.4. Schnorr Signatures: There are many variants of the DSA signature scheme based on discrete logarithms. A particularly interesting one is that of Schnorr signatures. We present the algorithm in the general case and allow the reader to work out the differences between the elliptic curve and finite field variants.

Suppose G is a public finite abelian group generated by an element g of prime order q. The public/private key pairs are just the same as in DSA, namely

- The private key is an integer x in the range $0 < x < q$.
- The public key is the element $h \leftarrow g^x$.

Signing: To sign a message m using the Schnorr signature algorithm the signer performs the following steps:

(1) $k \leftarrow \mathbb{Z}/q\mathbb{Z}$.
(2) $r \leftarrow g^k$.
(3) $e \leftarrow H(m\|r)$.
(4) $s \leftarrow k + x \cdot e \pmod q$.

The signature is then given by the pair (e, s). Notice how the hash function depends both on the message and the ephemeral public key r; we will see this is crucial in order to establish the security results below. In addition, notice that the signing equation is easier than that used for DSA, as we do not require a modular inversion modulo q.

Verification: The verification step is very simple:

- $r \leftarrow g^s \cdot h^{-e}$.
- The signature is accepted if and only if $e = H(m\|r)$.

Schnorr Signature Example: As an example of Schnorr signatures in a finite field we take the domain parameters $q = 101$, $p = 607$ and $g = 601$. As the public/private key pair we assume $x \leftarrow 3$ and $h \leftarrow g^x \pmod p = 391$. Then to sign a message we generate an ephemeral key, let's take $k \leftarrow 65$, and compute $r \leftarrow g^k \pmod p = 223$. We now need to compute the hash value $e \leftarrow h(m\|r) \pmod q$. Let us assume that we compute $e = 93$; then the second component of the signature is given by

$$s \leftarrow k + x \cdot e \pmod q = 65 + 3 \cdot 93 \pmod{101} = 41.$$

We leave it to the reader to check that the signature (e, s) verifies, i.e. that the verifier recovers the same value of r.

Schnorr Authentication Protocols: Schnorr signatures have been suggested for use in challenge response mechanisms in smart cards since the response part of the signature (the value of s) is particularly easy to evaluate because it only requires the computation of a single modular multiplication and a single modular addition. No matter what group we choose this final phase only requires arithmetic modulo a relatively small prime number.

To see how one uses Schnorr signatures in a challenge response situation we give the following scenario. You wish to use a smart card to authenticate yourself to a building or ATM machine. The card reader has a copy of your public key h, whilst the card has a copy of your private key x. Whilst you are walking around the card is generating commitments, which are ephemeral public keys of the form $r = g^k$.

When you place your card into the card reader the card transmits to the reader the value of one of these precomputed commitments. The card reader then responds with a challenge message e. Your card then only needs to compute

$$s = k + x \cdot e \pmod{q},$$

and transmit it to the reader, which then verifies the 'signature' by checking whether

$$g^s = r \cdot h^e.$$

Notice that the only online computations needed by the card are the computations of the values of e and s, which are both easy to perform.

In more detail, if we let C denote the card and R denote the card reader then we have

$$\begin{aligned} C &\longrightarrow R : r = g^k, \\ R &\longrightarrow C : e, \\ C &\longrightarrow R : s = k + xe \pmod{q}. \end{aligned}$$

The point of the initial commitment is to stop either the challenge being concocted so as to reveal your private key, or your response being concocted so as to fool the reader. A three-phase protocol consisting of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is a common form of authentication protocol, and we shall see more protocols of this nature when we discuss zero-knowledge proofs in Chapter 21.

Schnorr Signature Security: We shall now prove that Schnorr signatures are EUF-CMA secure in the random oracle model. The proof uses something called the forking lemma, which we present without proof; see the paper of Pointcheval and Stern mentioned at the end of this chapter.

Lemma 16.11 (Forking Lemma)**.** *Let A be a randomized algorithm with inputs $(x, h_1, \ldots, h_q, r)$, where r is the randomness used by A drawn from a distribution R, and the values $h_1, \ldots, h_q$ are selected from a set $\mathcal{H}$ uniformly at random.*

Assume that A outputs a pair (t, y). Let ϵ_A be the probability that the value t output by A is in the range $[1, \ldots, q]$. Define the algorithm B in Algorithm 16.7, which has input x, and let ϵ_B denote the probability that B outputs a non-zero tuple. Then

$$\epsilon_B \geq \frac{\epsilon_A^2}{q} - \frac{\epsilon_A}{|\mathcal{H}|}.$$

How we think about (and use) the A and B of the Lemma is as follows. Algorithm A is assumed to be running in the random oracle model, and the h_i values are the responses it receives to its random oracle queries. There is a special query which A uses in producing its answer y; this is called the *critical query*, and it is denoted by t in Algorithm 16.7. If $t = 0$ then algorithm A is not successful. We now run A again, with the same random tape and the same random oracle, up until the tth query. At this point the random oracle changes. At this point, which recall was the

Algorithm 16.7: Forking algorithm B

$r \leftarrow R$.
$h_1, \ldots, h_q \leftarrow \mathcal{H}$.
$(t, y) \leftarrow A(x, h_1, \ldots, h_q, r)$.
if $t = 0$ **then return** $(0, 0)$.
$h'_t, \ldots, h'_q \leftarrow \mathcal{H}$.
$(t', y') \leftarrow A(x, h_1, \ldots, h_{t-1}, h'_t, \ldots, h'_q, r)$.
if $t \neq t'$ or $h_t = h'_t$ **then return** $(0, 0)$.
return (y, y').

critical query, the second execution of A "forks" down another path (giving the lemma its name). The lemma tells us a lower bound on the probability that the two runs of A result in the same value for the critical query, assuming it is distinct.

The lemma is important in analysing signature schemes which are of the form *commit*, *challenge*, *response*. To realize such signatures we use the following notation. To sign a message

- The signer produces a (possibly empty) commitment σ_1 (the commitment).
- The signer computes $e = H(\sigma_1 \| m)$ (the challenge).
- The signer computes σ_2 which is the 'signature' on σ_1 and e (the response).

We label the output of the signature schemes as $(\sigma_1, H(\sigma_1 \| m), \sigma_2)$ so as to keep track of the exact hash query; DSA, EC-DSA and Schnorr signatures are all of this form:

- DSA : $\sigma_1 = \emptyset$, $e = H(m)$, $\sigma_2 = (r, (e + x \cdot r)/k \pmod q)$ where $r = \left(g^k \pmod p\right) \pmod q$.
- EC-DSA : $\sigma_1 = \emptyset$, $e = H(m)$, $\sigma_2 = (r, (e + x \cdot r)/k \pmod q)$, where $r = x\text{-coord}([k]G)$.
- Schnorr signatures: $\sigma_1 = g^k$, $e = H(\sigma_1 \| m)$ $\sigma_2 = x \cdot e + k \pmod q$.

In all of these schemes the hash function is assumed to have codomain equal to $\mathbb{F}_q$.

Recall that in the random oracle model the hash function is allowed to be cooked up by the algorithm B to do whatever it likes. Suppose an adversary A can produce an existential forgery on a message m with non-negligible probability in the random oracle model. Hence, the output of the adversary is

$$(m, \sigma_1, e, \sigma_2).$$

We can assume that the adversary makes the critical hash query for the forged message, $e = H(\sigma_1 \| m)$, since otherwise we can make the query for the adversary ourselves.

Algorithm B now runs the adversary A again, just as in the forking lemma, with the same random tape and the modified random oracle. Up until the critical query, the hash queries were answered the same way as before, so we have that the execution of A in both runs is identical up until the critical query. If Algorithm 16.7 is successful this means the two executions output two tuples

$$y = (m, \sigma_1, e, \sigma_2) \text{ and } y' = (m', \sigma'_1, e', \sigma'_2),$$

where e and e' are the outputs from the critical query, but the inputs to this query are *the same*. In other words we have $m = m'$ and $\sigma_1 = \sigma'_1$. This last equation does not give us anything in the case of DSA or EC-DSA, since in those cases σ_1 is always equal to $\emptyset$. However, for Schnorr signatures we obtain something useful, since we find $g^k = g^{k'}$, where k and k' are the underlying ephemeral keys of the two signatures. Note that A might not even know k and k' in running her attack code. This allows us to deduce $k = k'$, and hence to recover the secret key x from the equation

$$x = \frac{e - e'}{\sigma_2 - \sigma'_2}.$$

Notice that the denominator here is non-zero by assumption. This is the basic idea behind the proof of the following theorem.

Theorem 16.12. *In the random oracle model let A denote a* EUF-CMA *adversary against Schnorr signatures with advantage ϵ, making q_H queries to its hash function H. Then there is an adversary B against the discrete logarithm problem with advantage ϵ' such that*

$$\epsilon' \geq \frac{\epsilon^2}{q} - \frac{\epsilon}{q}.$$

Proof. Algorithm B has as input g and $h = g^x$ and it wishes to find x. The value h will be used as the public key by algorithm A. We first 'package' A into an algorithm A' which does not require access to a signing oracle as follows. The algorithm A' will take as input $(h, h_1, \ldots, h_q, r)$, where r is an entry from the set of possible random tapes of algorithm A. When A makes a signing query on the message m then algorithm A' executes the following steps:

- Take the next hash query value input to A', let this be value h_i.
- $s \leftarrow \mathbb{Z}/q\mathbb{Z}$.
- $r \leftarrow g^s/h^e$.
- Define $H(m\|r) = h_i$. If this value has already been defined then pick another value of s.

The hash function queries are handled in the usual way, using the inputs $h_1, \ldots, h_q$. If A does not terminate with a forged signature then A' output $(0, 0)$, otherwise it outputs (t, y) where t is the index of the critical hash query and

$$y = (m, \sigma_1, e, \sigma_2),$$

with $h_t = e$. Algorithm A' is now an algorithm which we can use in the forking lemma. This gives us an algorithm B which will produce two values y and y', from which we can recover x via the above method. □

16.5.5. Nyberg–Rueppel Signatures: What happens when we want to sign a general message which is itself quite short? It may turn out that the signature could be longer than the message. Recall that RSA can be used either as a scheme with appendix or as a scheme with message recovery. So far none of our discrete-logarithm-based schemes can be used with message recovery. We end this section by giving an example scheme which does have the message recovery property, called the Nyberg–Rueppel signature scheme, which is based on discrete logarithms in some public finite abelian group G.

Many signature schemes with message recovery require a public redundancy function R. This function maps actual messages over to the data which is actually signed. This acts rather like a hash function does in the schemes based on signatures with appendix. However, unlike a hash function the redundancy function must be easy to invert. As a simple example we could take R to be the function

$$R : \begin{cases} \{0,1\}^{n/2} \longrightarrow \{0,1\}^n \\ m \longmapsto m\|m. \end{cases}$$

We assume that the codomain of R can be embedded into the group G. In our description we shall use the integers modulo p, i.e. $G = \mathbb{F}_p^*$, and as usual we assume that a large prime q divides $p - 1$ and that g is a generator of the subgroup of order q. Once again the public/private key pair is given as a discrete logarithm problem $\mathfrak{pk} \leftarrow h = g^x$.

Signing: Nyberg–Rueppel signatures are then produced as follows:

(1) $k \leftarrow \mathbb{Z}/q\mathbb{Z}$.
(2) $r \leftarrow g^k \pmod p$.
(3) $e \leftarrow R(m) \cdot r \pmod p$.
(4) $s \leftarrow x \cdot e + k \pmod q$.

The signature is then the pair (e, s).

Verification and Recovery: From the pair (e, s) and the public key h we need to

- Verify that the signature comes from the user with public key h,
- Recover the message m from the pair (e, s).

This is performed as follows:

(1) $u_1 \leftarrow g^s \cdot h^{-e} = g^{s-e\cdot x} = g^k \pmod p$.
(2) $u_2 \leftarrow e/u_1 \pmod p$.
(3) Verify that u_2 lies in the range of the redundancy function, e.g. we must have $u_2 = R(m) = m\|m$. If this does not hold then reject the signature.
(4) Recover the message $m = R^{-1}(u_2)$ and accept the signature.

Example: As an example we take the domain parameters $q = 101$, $p = 607$, $g = 601$, and as the redundancy function we take $R(m) = m + 2^4 \cdot m$, where a message m must lie in $[0, \ldots, 15]$. As the public/private key pair we assume $x \leftarrow 3$ and $h \leftarrow g^x \pmod p = 391$. To sign the message $m = 12$ we compute an ephemeral key $k \leftarrow 45$ and $r \leftarrow g^k \pmod p = 143$. Since $R(m) = m + 2^4 \cdot m$ we have $R(m) = 204$. We then compute $e \leftarrow R(m) \cdot r \pmod p = 36$, $s \leftarrow x \cdot e + k \pmod q = 52$. The signature is then the pair $(e, s) = (36, 52)$.

We now show how this signature is verified and the message recovered. We first compute $u_1 = g^s \cdot h^{-e} = 143$. Notice how the verifier has computed u_1 to be the same as the value of r computed by the signer. The verifier now computes $u_2 = e/u_1 \pmod p = 204$. The verifier now checks that $u_2 = 204$ is of the form $m + 2^4 m$ for some value of $m \in [0, \ldots, 15]$. We see that u_2 is of this form and so the signature is valid. The message is then recovered by solving for m in $m + 2^4 m = 204$, from which we obtain $m = 12$.

16.6. Schemes Avoiding Random Oracles

In the previous sections we looked at signature and encryption schemes which can be proved secure in the so-called 'random oracle model'. A proof in the random oracle model only provides *evidence* that a scheme may be secure in the real world, it does not guarantee security in the real world. We can interpret a proof in the random oracle model as saying that if an adversary against the real-world scheme exists then that adversary must make use of the specific hash function employed.

In this section we sketch recent ways in which researchers have tried to construct signature and encryption algorithms which do not depend on the random oracle model, i.e. schemes in the standard model. We shall only consider schemes which are practical, and we shall only sketch the proof ideas. Readers interested in the details of proofs or in other schemes should consult the extensive literature in this area.

What we shall see is that whilst quite natural encryption algorithms can be proved secure without the need for random oracles, the situation is quite different for signature algorithms. This should not be surprising since signature algorithms make extensive use of hash functions for their security. Hence, we should expect that they impose stricter restraints on such hash functions, which may not actually be true in the real world.

16.6.1. The Cramer–Shoup Encryption Scheme: Unlike the case of signature schemes in the standard model, for encryption algorithms one can produce provably secure systems which are practical and close to those used in 'real life'. The Cramer–Shoup encryption scheme requires as domain parameters a finite abelian group G of prime order q. In addition we require a one-way family of hash functions. This is a family $\{H_i\}$ of hash functions for which it is hard for an adversary to choose an input x, then to draw a random hash function H_i, and then to find a different input y so that

$$H_i(x) = H_i(y).$$

Key Generation: A public key in the Cramer–Shoup scheme is chosen as follows. First the following random elements are selected

$$g_1, g_2 \leftarrow G,$$
$$x_1, x_2, y_1, y_2, z \leftarrow \mathbb{Z}/q\mathbb{Z}.$$

The user then computes the following elements

$$c \leftarrow {g_1}^{x_1} \cdot {g_2}^{x_2},$$
$$d \leftarrow {g_1}^{y_1} \cdot {g_2}^{y_2},$$
$$h \leftarrow {g_1}^{z}.$$

The user finally chooses a hash function H from the universal one-way family of hash functions and outputs the public key $\mathfrak{pk} \leftarrow (g_1, g_2, c, d, h, H)$, whilst keeping secret the private key $\mathfrak{sk} \leftarrow (x_1, x_2, y_1, y_2, z)$.

Encryption: The encryption algorithm proceeds as follows, which is very similar to ElGamal encryption. The message m is considered as an element of G, and encryption proceeds as follows:

$$r \leftarrow \mathbb{Z}/q\mathbb{Z},$$
$$u_1 \leftarrow {g_1}^{r},$$
$$u_2 \leftarrow {g_2}^{r},$$
$$e \leftarrow m \cdot h^{r},$$
$$\alpha \leftarrow H(u_1 \| u_2 \| e),$$
$$v \leftarrow c^{r} \cdot d^{r\alpha}.$$

The ciphertext is then the quadruple (u_1, u_2, e, v).

Decryption: On receiving this ciphertext the owner of the private key can recover the message as follows: First they compute $\alpha \leftarrow H(u_1 \| u_2 \| e)$ and test whether

$${u_1}^{x_1 + y_1\alpha} \cdot {u_2}^{x_2 + y_2\alpha} = v.$$

If this equation does not hold then the ciphertext should be rejected. If this equation holds then the receiver can decrypt the ciphertext by computing

$$m \leftarrow \frac{e}{{u_1}^{z}}.$$

Notice that, whilst very similar to ElGamal encryption, the Cramer–Shoup encryption scheme is much less efficient. Hence, whilst provably secure it is not used much in practice.

Security: To show that the scheme is provably secure, under the assumption that the DDH problem is hard and that H is chosen from a universal one-way family of hash functions, we assume we have an adversary A against the scheme and show how to use A in another algorithm B which tries to solve the DDH problem.

One way to phrase the DDH problem is as follows: Given $(g_1, g_2, u_1, u_2) \in G$ determine whether this quadruple is a random quadruple or we have $u_1 = {g_1}^{r}$ and $u_2 = {g_2}^{r}$ for some value of $r \in \mathbb{Z}/q\mathbb{Z}$. So algorithm B will take as input a quadruple $(g_1, g_2, u_1, u_2) \in G$ and try to determine whether this is a random quadruple or a quadruple related to the Diffie–Hellman problem.

Algorithm B first needs to choose a public key, which it does in a non-standard way, by first selecting the random elements

$$x_1, x_2, y_1, y_2, z_1, z_2 \in \mathbb{Z}/q\mathbb{Z}.$$

Algorithm B_A then computes the following elements

$$c \leftarrow {g_1}^{x_1} \cdot {g_2}^{x_2},$$
$$d \leftarrow {g_1}^{y_1} \cdot {g_2}^{y_2},$$
$$h \leftarrow {g_1}^{z_1} \cdot {g_2}^{z_2}.$$

Finally B chooses a hash function H from the universal one-way family of hash functions and outputs the public key $\mathfrak{pk} \leftarrow (g_1, g_2, c, d, h, H)$. Notice that the part of the public key corresponding to h has been chosen differently than in the real scheme, but that algorithm A will not be able to detect this change. Algorithm B now runs algorithm A, responding to decryption queries of (u_1', u_2', e', v') by computing

$$m \leftarrow \frac{e'}{{u_1'}^{z_1} {u_2'}^{z_2}},$$

after performing the standard check on validity of the ciphertext.

At some point A calls its $\mathcal{O}_{\mathsf{LR}}$ oracle on the two plaintexts m_0 and m_1. Algorithm B chooses a bit b at random and computes the target ciphertext as

$$e \leftarrow m_b \cdot \left({u_1}^{z_1} \cdot {u_2}^{z_2}\right),$$
$$\alpha \leftarrow H(u_1 \| u_2 \| e),$$
$$v \leftarrow {u_1}^{x_1 + y_1 \alpha} \cdot {u_2}^{x_2 + y_2 \alpha}.$$

The target ciphertext is then the quadruple (u_1, u_2, e, v). Notice that when the input to B is a legitimate DDH quadruple then the target ciphertext will be a valid encryption, but when the input to B is not a legitimate DDH quadruple then the target ciphertext is highly likely to be an invalid ciphertext. This target ciphertext is then returned to the adversary A.

If the adversary outputs the correct value of b then we suspect that the input to B is a valid DDH quadruple, whilst if the output is wrong then we suspect that the input to B is not valid. This produces a statistical test to detect whether the input to B was valid or not. By repeating this test a number of times we can produce as accurate a statistical test as we want.

Note that the above is only a sketch. We need to show that the view of the adversary A in the above game is no different from that in a real attack on the system, otherwise A would know something was not correct. For example we need to show that the responses B makes to the decryption queries of A cannot be distinguished from a true decryption oracle. For further details one should consult the full proof in the paper mentioned in the Further Reading section.

16.6.2. Cramer–Shoup Signatures: We have already remarked that signature schemes which are provably secure, without the random oracle model, are hard to come by. They also appear somewhat contrived compared with the schemes such as RSA-PSS, DSA or Schnorr signatures which are used in real life. The first such provably secure signature scheme in the standard model was by Goldwasser, Micali and Rivest. This was however not very practical as it relied on messages being associated with leaves of a binary tree, and each node in the tree needed to be authenticated with respect to its parent. This made the resulting scheme far too slow.

However, even with today's knowledge the removal of the use of random oracles comes at a significant price. We need to make stronger intractability assumptions than we have otherwise made. In this section we introduce a new RSA-based problem called the Flexible RSA Problem. This is a potentially easier problem than ones we have met before, hence the assumption that the problem is hard is a much stronger assumption than before. The *strong RSA assumption* is the assumption that the Flexible RSA Problem is hard.

Definition 16.13 (Flexible RSA Problem)**.** *Given an RSA modulus $N = p \cdot q$ and a random $c \in (\mathbb{Z}/N\mathbb{Z})^*$ find $e > 1$ and $m \in (\mathbb{Z}/N\mathbb{Z})^*$ such that*

$$m^e = c.$$

Clearly if we can solve the RSA problem then we can solve the Flexible RSA Problem. This means that the strong RSA assumption is a stronger intractability assumption than the standard RSA assumption, in that it is conceivable that in the future we may be able to solve the Flexible RSA Problem but not the traditional RSA problem. However, at present we conjecture that both problems are equally hard.

The Cramer–Shoup signature scheme is based on the strong RSA assumption and is provably secure, without the need for the random oracle model. The main efficiency issue with the scheme is that the signer needs to generate a new prime number for each signature produced, which can be rather costly. In our discussion below we shall assume H is a 'standard' hash function which outputs bit strings of 256 bits, which we interpret as 256-bit integers as usual.

Key Generation: To generate the public key, we create an RSA modulus N which is the product of two "safe" primes p and q, i.e. $p \leftarrow 2 \cdot p' + 1$ and $q \leftarrow 2 \cdot q' + 1$ where p' and q' are primes. We also choose two random elements

$$h, x \in Q_N,$$

where, as usual, Q_N is the set of quadratic residues modulo N. We also create a random 256-bit prime e'. The public key consists of

$$(N, h, x, e')$$

and the private key is the factors p and q.

Signing: To sign a message the signer generates another 256-bit prime number e and another random element $y' \in Q_N$. Since they know the factors of N, the signer can compute the solution y to the equation

$$y = \left(x \cdot h^{H(x')}\right)^{1/e} \pmod{N},$$

where x' satisfies

$$x' = y'^{e'} \cdot h^{-H(m)}.$$

The output of the signer is (e, y, y').

Verification: To verify a message the verifier first checks that e' is an odd number satisfying $e \neq e'$. Then the verifier computes

$$x' \leftarrow y'^{e'} \cdot h^{-H(m)}$$

and then checks that

$$x = y^e \cdot h^{-H(x')}.$$

Security: On the assumption that H is a collision resistant hash function and the Flexible RSA Problem is hard, one can prove that the above scheme is secure against active adversaries. We sketch the most important part of the proof, but the full details are left to the interested reader to look up in the paper mentioned at the end of this chapter.

Assume the adversary makes t queries to a signing oracle. We want to use the adversary A to create an algorithm B to break the strong RSA assumption for the modulus N. Before setting up the public key for input to algorithm A, the algorithm B first decides on what prime values e_i it will output in the signature queries. Then, having knowledge of the e_i, the algorithm B concocts values for the h and x in the public key, so that it always knows the e_ith root of h and x.

Thus when given a signing query for a message m_i, algorithm B can then compute a valid signature, without knowing the factorization of N, by generating $y_i' \in Q_N$ at random and then computing

$$x_i' \leftarrow y_i'^{e'} \cdot h^{-H(m_i)} \pmod{N}$$

and then

$$y_i \leftarrow x^{1/e_i} \cdot (h^{1/e_i})^{H(x_i')} \pmod{N},$$

the signature being given by

$$(m_i, y_i, y_i').$$

The above basic signing simulation is modified in the full proof, depending on what type of forgery algorithm A is producing. But the basic idea is that B creates a public key to enable it to respond to every signing query in a valid way.

Chapter Summary

- ElGamal encryption is a system based on the difficulty of the Diffie–Hellman problem (DHP).
- Paillier encryption is a scheme which is based on the Composite Decision Residuosity Problem (CDRP).
- The random oracle model is a computational model used in provable security. A proof in the random oracle model does not mean the system is secure in the real world, it only provides *evidence* that it *may* be secure.
- In the random oracle model one can prove that the ubiquitous RSA encryption method, namely RSA-OAEP, is secure.
- We presented the KEM-DEM paradigm for designing public key encryption schemes.
- We gave the RSA-KEM (resp. DHIES-KEM) and showed why it is secure, assuming the RSA (resp. Gap Diffie–Hellman) problem is hard.
- In the random oracle model the two main RSA based signature schemes used in 'real life' are also secure, namely RSA-FDH and RSA-PSS.
- DSA is a signature algorithm based on discrete logarithms; it has reduced bandwidth compared with RSA but is slower. EC-DSA is the elliptic curve variant of DSA; it has reduced bandwidth and greater efficiency compared to DSA.
- In the random oracle model one can use the forking lemma to show that the Schnorr signature scheme is secure.
- The Flexible RSA Problem is a natural weakening of the standard RSA problem.
- The Cramer–Shoup encryption scheme is provably secure, without the random oracle model, assuming the DDH problem is hard. It is around three times slower than ElGamal encryption.

Further Reading

The paper by Cramer and Shoup on public key encryption presents the basics of hybrid encryption in great detail, as well as the scheme for public key encryption without random oracles. The other paper by Cramer and Shoup presents their signature scheme. The DHIES scheme was first presented in the paper by Abdalla et al. A good paper to look at for various KEM constructions is that by Dent. The full proof of the security of RSA-OAEP is given in the paper of Fujisaki et al.

A good description of the forking lemma and its applications is given in the article of Pointcheval and Stern. The random oracle model and a number of applications including RSA-FDH and RSA-PSS are given in the papers of Bellare and Rogaway.

M. Abdalla, M. Bellare and P. Rogaway. *DHAES: An encryption scheme based on the Diffie–Hellman problem*. Submission to IEEE P1363a standard.

M. Bellare and P. Rogaway. *Random oracles are practical: a paradigm for designing efficient protocols*. In Proc. 1st Annual Conf. on Comp. and Comms. Security, 62–73, ACM, 1993.

M. Bellare and P. Rogaway. *The exact security of digital signatures – How to sign with RSA and Rabin*. In Advances in Cryptology – Eurocrypt 1996. LNCS 1070, 399–416, Springer, 1996.

R. Cramer and V. Shoup. *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack*. SIAM Journal on Computing **33**, 167–226, 2003.

R. Cramer and V. Shoup. *Signature schemes based on the strong RSA assumption*. ACM Transactions on Information and Systems Security, **3**, 161–185, 2000.

A. Dent. *A designer's guide to KEMs*. In Cryptography and Coding – 2003, LNCS 2898, 133–151, Springer, 2003.

E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. *RSA-OAEP is secure under the RSA assumption*. In Advances in Cryptology – Crypto 2001, LNCS 2139, 260–274, Springer, 2001.

D. Pointcheval and J. Stern. *Security arguments for digital signatures and blind signatures*. J. Cryptology, **13**, 361–396, 2000.

CHAPTER 17

Cryptography Based on Really Hard Problems

Chapter Goals

- To introduce the concepts of complexity theory needed to study cryptography.
- To understand why complexity theory on its own cannot lead to secure cryptographic systems.
- To introduce the idea of random self-reductions.
- To explain the Merkle–Hellman system and why it is weak.
- To sketch the idea behind worst-case to average-case reductions for lattices.
- To introduce the Learning with Errors problem, and the Ring-Learning with Errors problem, and to describe a public key encryption scheme based on them.
- To sketch how to extend this encryption scheme to produce a fully homomorphic encryption scheme.

17.1. Cryptography and Complexity Theory

Up until now we have looked at basing cryptography on problems which are *believed* to be hard, e.g. that AES is a PRF, that factoring a product of large primes is hard, that finding discrete logarithms is hard. But there is no underlying *reason* why these problems should be hard. Computer Science gives us a whole theory of categorizing hard problems, called *complexity theory*. Yet none of our hard problems appear to be what a complexity theorist would call hard. Indeed, in comparison to what complexity theorists discuss, factoring and discrete logarithms are comparatively easy.

We are going to need to recap some basic complexity theory, most of which one can find in a basic computer science undergraduate curriculum, or by reading the book by Goldreich in the Further Reading section of this chapter.

17.1.1. Decision and Search Problems: A decision problem $\mathcal{DP}$ is a problem with a yes/no answer, which has inputs (called instances) ι coded in some way (for example as a binary string of some given size n). Often one has a certain set S of instances in mind and one is asking "Is $\iota \in S$?". For example one could have

- ι is the encoding of an integer, and S is the set of all primes. Hence the decision problem is: Given an integer, N, say whether it is prime or not.
- ι is the encoding of a graph, and S is the subset of all graphs which are colourable using k colours only. Recall that a graph consisting of vertices V and edges E is colourable by k colours if one can assign a colour (or label) to each vertex so that no two vertices connected by an edge share the same colour. Hence the decision problem is: Given a graph, G, say whether it is colourable using only k colours.

Whilst we have restricted ourselves to decision problems, one can often turn a standard computational problem into a decision problem. As an example of this consider the cryptographically important knapsack problem.

Definition 17.1 (Decision Knapsack Problem)**.** *Given a pile of n items, with different weights w_i, is it possible to put items into a knapsack to make a specific weight S? In other words, do there exist $b_i \in \{0,1\}$ such that $S = b_1 \cdot w_1 + b_2 \cdot w_2 + \cdots + b_n \cdot w_n$?*

As stated above the knapsack problem is a decision problem but we could ask for an algorithm to actually *search* for the values b_i.

Definition 17.2 (Knapsack (Search) Problem)**.** *Given a pile of n items, with different weights w_i, is it possible to put items into a knapsack to make a specific weight S? If so can one find the $b_i \in \{0,1\}$ such that $S = b_1 \cdot w_1 + b_2 \cdot w_2 + \cdots + b_n \cdot w_n$? We assume only one such assignment of weights is possible.*

Note that the time taken to solve either of these variants of the knapsack problem seems to grow in the worst case as an exponential function of the number of weights. We can turn an oracle for the decision knapsack problem into one for the knapsack problem proper. To see this consider Algorithm 17.1 which assumes an oracle $O(w_1, ..., w_n, S)$ for the decision knapsack problem.

Algorithm 17.1: Knapsack algorithm, assuming a decision knapsack oracle

```
if O(w_1, ..., w_n, S) =false then return false.
T ← S.
b_1 ← 0, ..., b_n ← 0.
for i = 1 to n do
    if T = 0 then return (b_1, ..., b_n).
    if O(w_{i+1}, ..., w_n, T − w_i) =true then
        T ← T − w_i.
        b_i ← 1.
```

A decision problem $\mathcal{DP}$, characterized by a set S, is said to lie in complexity class $\mathcal{P}$ if there is an algorithm which takes any instance ι, and decides whether or not $\iota \in S$ in polynomial time. We measure time in terms of bit operations and polynomial time means the number of bit operations is bounded by some polynomial function of the input size of the instance ι.

The problems which lie in complexity class $\mathcal{P}$ are those for which we have an "efficient" solution algorithm[1]. In other words things in complexity class $\mathcal{P}$ are those things which are believed to be easy to compute. For example.

- Given integers x, y and z do we have $z = x \cdot y$, i.e. is multiplication easy?
- Given a ciphertext c, a key k and a plaintext m, is c the encryption of m under your favourite encryption algorithm?

Of course in the last example I have assumed your favourite encryption algorithm has an encryption/decryption algorithm which runs in polynomial time. If your favourite encryption algorithm is not of this form, then one must really ask how have you read so far in this book?

17.1.2. The class $\mathcal{NP}$: A decision problem lies in complexity class $\mathcal{NP}$, called non-deterministic polynomial time, if for every instance for which the answer is *yes*, there exists a witness for this which can be checked in polynomial time. If the answer is *no* we do not assume the algorithm terminates, but if it does it must do so by answering *no*. One should think of the witness as a proof that the instance ι lies in the subset S. Examples include

- The problem "Is N composite?" lies in $\mathcal{NP}$ as one can give a non-trivial prime factor as a witness. This witness can be checked in polynomial time since division can be performed in polynomial time.

[1] Of course efficiency in practice may be different as the polynomial bounding the run time could have high degree.

- The problem "Is G k-colourable?" lies in $\mathcal{NP}$ since as a witness one can give the colouring.
- The problem "Does this knapsack problem have a solution?" lies in $\mathcal{NP}$ since as a witness we can give the values b_i.

Note that in these examples we *do not* assume the witness itself can be *computed* in polynomial time, only that the witness can be *checked* in polynomial time. Note that we trivially have

$$\mathcal{P} \subseteq \mathcal{NP}.$$

The main open problem in theoretical computer science is the question: Does $P = \mathcal{NP}$? Most people believe in the following conjecture.

Conjecture 17.3.

$$P \neq \mathcal{NP}.$$

The class co-$\mathcal{NP}$ is the set of problems for which a witness exists for every instance with a *no* response which can be checked in polynomial time. It is conjectured that $\mathcal{NP} \neq$ co-$\mathcal{NP}$, and if a problem lies in $\mathcal{NP} \cap$co-$\mathcal{NP}$ then this is seen as evidence that the problem cannot be $\mathcal{NP}$-complete. For example the problem of determining whether a number n has a prime factor less than m can be shown to lie in $\mathcal{NP} \cap$ co-$\mathcal{NP}$, thus we do not think that this problem is $\mathcal{NP}$-complete. In other words, factoring is not that hard a problem from a complexity-theoretic point of view.

One can consider trying to see how small a witness for being in class $\mathcal{NP}$ can be. For example consider the problem COMPOSITES. Namely, given $N \in \mathbb{Z}$ determine whether N is composite. As we remarked earlier this clearly lies in class $\mathcal{NP}$. But a number N can be proved composite in the following ways:

- Giving a factor. In this case the size of the witness is $O(\log N)$.
- Giving a Miller–Rabin witness a. Now, assuming the Generalized Riemann Hypothesis (GRH) the size of a witness can be bounded by $O(\log \log N)$ since we have $a \leq O((\log N)^2)$.

A decision problem $\mathcal{DP}$, in the class $\mathcal{NP}$, is said to be $\mathcal{NP}$-complete if every other problem in class $\mathcal{NP}$ can be reduced to this problem in polynomial time. In other words we have: if $\mathcal{DP}$ is $\mathcal{NP}$-complete then

$$\mathcal{DP} \in \mathcal{P} \text{ implies } \mathcal{P} = \mathcal{NP}.$$

In some sense the $\mathcal{NP}$-complete problems are the hardest problems for which it is feasible to ask for a solution. There are a huge number of $\mathcal{NP}$-complete problems of which two will interest us:

- the 3-colouring problem,
- the knapsack problem.

A problem $\mathcal{DP}$ is called $\mathcal{NP}$-hard if we have that $\mathcal{DP} \in \mathcal{P}$ implies $\mathcal{P} = \mathcal{NP}$, but we do not know whether $\mathcal{DP} \in \mathcal{NP}$. Thus an $\mathcal{NP}$-hard problem is a problem which is as hard as every problem in $\mathcal{NP}$, and could be even harder.

17.1.3. Average vs Worst-Case Complexity: It is a widely held view that all the standard hard problems on which cryptography is based, e.g. factoring, discrete logarithms etc., are not equivalent to an $\mathcal{NP}$-complete problem even though they lie in class $\mathcal{NP}$. From this we can conclude that factoring, discrete logarithms etc. are not very difficult problems at all, at least not compared with the knapsack problem or the 3-colouring problem. So why do we not use $\mathcal{NP}$-complete problems on which to base our cryptographic schemes? These are, after all, a well-studied set of problems for which we do not expect there ever to be an efficient solution.

However, this approach has had a bad track record, as we shall show later when we consider the knapsack-based system of Merkle and Hellman. For now we simply mention that the theory of $\mathcal{NP}$-completeness is about worst case complexity. But for cryptography we want a problem which, for suitably chosen parameters, is hard on average. It turns out that the knapsack problems that have in the past been proposed for use in cryptography are always "average" and efficient algorithms can always be found to solve them.

We illustrate this difference between hard and average problems using the k-colouring problem, when $k = 3$. Although determining whether a graph is 3-colourable is in general (in the worst case) $\mathcal{NP}$-complete, it is very easy on average. This is because the average graph, no matter how large it is, will not be 3-colourable. In fact, for almost all input graphs the following algorithm will terminate saying that the graph is not 3-colourable in a constant number of iterations.

- Take a graph G and order the vertices in any order $v_1, \ldots, v_t$.
- Call the colours $\{1, 2, 3\}$.
- Now traverse the graph in the order of the vertices just decided.
- On visiting a new vertex select the smallest possible colour (i.e. one from the set $\{1, 2, 3\}$) which does not appear as the colour of an adjacent vertex. Select this as the colour of the current vertex.
- If you get stuck (i.e. no such colour exists) traverse back up the graph to the most recently coloured vertex and use the next colour available, then continue down again.
- If at any point you run out of colours for the first vertex then terminate and say the graph is *not* 3-colourable.
- If you are able to colour the last vertex then terminate and output that the graph *is* 3-colourable.

The interesting thing about the above algorithm is that it can be shown that for a *random* graph of t vertices the average number of vertices travelled in the algorithm is less than 197 regardless of the number of vertices t in the graph.

Now consider the case of factoring. Factoring is actually easy on average, since if you give me a large number I can usually find a factor quite quickly. Indeed for fifty percent of large numbers I can find a factor by just examining one digit of the number; after all fifty percent of large numbers are even! When we consider factoring in cryptography we implicitly mean that we construct a *hard* instance by multiplying two large primes together. So could such an example work for $\mathcal{NP}$-complete problems? That is can we quickly write down an instance which we know to be one of the hardest instances of such a problem? Think of the graph colouring example. How can we construct a graph which is hard to colour, but is 3-colourable? This seems really difficult. Later we shall see that for some hard problems related to lattices, we can choose parameters that make the average case as difficult as the worst case of another (related) hard lattice problem.

17.1.4. Random Self-reductions: So we know that we should select as hard a problem as possible on which to base our cryptographic security. For our public key schemes we pick parameters to avoid the obvious attacks based on factoring and discrete logarithms from Chapters 2 and 3. However, we also know that we do not actually base many cryptographic schemes on factoring or discrete logarithms per se. We actually use the RSA problem or the DDH problem. So we need to determine whether these problems are hard on average, and in what sense.

For example given an RSA modulus N and a public exponent e it might be hard to solve

$$c = m^e \pmod{N}$$

for a specific c in the worst case, but it could be easy on average for most values of c. It turns out that one can prove that problems such as the RSA problem for a fixed modulus N or DDH for a fixed group G are hard on average. The technique to do this is based on a random self-reduction from one given problem instance to another random problem instance of the *same* problem. This means that if we can solve the problem on average then we can solve the problem in the worst case, since if we had a worst-case problem, we could randomize it until we hit upon an easy average-case problem. Hence, the worst-case behaviour of the problem and the average-case behaviour of the problem must be similar.

Lemma 17.4. *The RSA problem is random self-reducible.*

PROOF. Suppose we are given c and are asked to solve $c = m^e \pmod N$, where the idea is that this is a "hard" problem instance. We reduce this to an "average" problem instance by choosing $s \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$ at random and setting $c' \leftarrow s^e c$. We then try to solve

$$c' = m'^e \pmod N.$$

If we are unsuccessful we choose another value of s until we hit the "average" type problem. If the average case was easy then we could solve $c' = m'^e \pmod N$ for m' and then set $m \leftarrow \frac{m'}{s}$ and terminate. □

One can also show that the DDH problem is random self-reducible, in the sense that testing whether $(A, B, C) = (g^a, g^b, g^c)$ is a valid Diffie–Hellman triple, i.e. whether $c = a \cdot b$, does not depend on the particular choices of a, b and c. To see this consider the related triple $(A', B', C') = (g^{a'}, g^{b'}, g^{c'}) = (A^u, B^v, C^{u \cdot v})$ for random u and v. Now if (A, B, C) is a valid Diffie–Hellman triplet then so is (A', B', C'), and vice versa.

One can show that the distribution of (A', B', C') will be uniform over all valid Diffie–Hellman triples if the original triple is a valid Diffie–Hellman triple, whilst the distribution will be uniform over all triples (and not just Diffie–Hellman ones) in the case where the original triple was not a valid Diffie–Hellman triple.

17.2. Knapsack-Based Cryptosystems

The idea of using known complexity-theoretic hard problems, as opposed to number-theoretic ones, has been around as long as public key cryptography. This topic has become more important in recent years as all of our existing number-theoretic constructions for public key cryptography would become broken if anyone built a quantum computer. Thus the hunt is on for secure and efficient public key primitives which are secure against any future quantum computer.

One of the earliest such public key cryptosystems was based on the knapsack or subset sum problem, which is $\mathcal{NP}$-complete. However it turns out that this knapsack-based scheme, and almost all others, can be shown to be insecure, as we shall now explain. The idea is to create two problem instances, a public one which is hard, which is believed to be a general knapsack problem, and a private problem which is easy. In addition there should be some private trapdoor information which transforms the hard problem into the easy one.

This is rather like the use of the RSA problem. It is hard to extract eth roots modulo a composite number, but easy to extract eth roots modulo a prime number. Knowing the trapdoor information, namely the factorization of the RSA modulus, allows us to transform the hard problem into the easy problem. However, the crucial difference is that whilst producing integers which are hard to factor is easy, it is difficult to produce knapsack problems which are hard on average. This is despite the general knapsack problem being considered harder than the general factorization problem.

Whilst the general knapsack problem is hard there is a particularly easy set of problems based on super-increasing knapsacks. A super-increasing knapsack problem is one where the weights are such that each one is greater than the sum of the preceding ones, i.e.

$$w_j > \sum_{i=1}^{j-1} w_i.$$

As an example one could take the set

$$\{2, 3, 6, 13, 27, 52\}$$

or one could take

$$\{1, 2, 4, 8, 16, 32, 64, \ldots\}.$$

Given a super-increasing knapsack problem, namely an ordered set of such super-increasing weights $\{w_1, \ldots, w_n\}$ and a target weight S, determining which weights to put in the sack is a linear operation, as can be seen from Algorithm 17.2.

Algorithm 17.2: Solving a super-increasing knapsack problem

```
for i = n downto 1 do
    if S >= w_i then
        b_i ← 1.
        S ← S − w_i.
    else
        b_i = 0.
if S = 0 then return (b_1, b_2, ..., b_n).
else return ("No Solution").
```

Key Generation: The Merkle–Hellman encryption scheme takes as a private key a super-increasing knapsack problem and from this creates (using a private transform) a so-called "hard knapsack" problem. This hard problem is then the public key. This transform is achieved by choosing two private integers N and M, such that

$$\gcd(N, M) = 1$$

and multiplying all values of the super-increasing sequence by $N \pmod M$. For example if we take as the private key

- the super-increasing knapsack $\{2, 3, 6, 13, 27, 52\}$,
- $N = 31$ and $M = 105$.

Then the associated public key is given by the "hard" knapsack

$$\{62, 93, 81, 88, 102, 37\}.$$

We then publish the hard knapsack problem as our public key, with the idea that only someone who knows N and M can transform back to the easy super-increasing knapsack.

Encryption: For Bob to encrypt a message to us, he first breaks the plaintext into blocks the size of the weight set. The ciphertext is then the sum of the weights where a bit is set. So for example if the message is given by

$$\text{Message} = 011000\ 110101\ 101110$$

Bob obtains, since our public knapsack is $\{62,\ 93,\ 81,\ 88,\ 102,\ 37\}$, that the ciphertext is

$$174,\ 280,\ 333,$$

since

- 011000 corresponds to $93 + 81 = 174$,
- 110101 corresponds to $62 + 93 + 88 + 37 = 280$,
- 101110 corresponds to $62 + 81 + 88 + 102 = 333$.

Decryption: The legitimate recipient knows the private key N, M and $\{2, 3, 6, 13, 27, 52\}$. Hence, by multiplying each ciphertext block by $N^{-1} \pmod M$ the hard knapsack is transformed into the easy knapsack problem. In our case $N^{-1} = 61 \pmod M$, and so the decryptor performs the operations

- $174 \cdot 61 = 9 = 3 + 6 = 011000$,
- $280 \cdot 61 = 70 = 2 + 3 + 13 + 52 = 110101$,
- $333 \cdot 61 = 48 = 2 + 6 + 13 + 27 = 101110$.

The final decoding is done using the simple easy knapsack,

$$\{2, 3, 6, 13, 27, 52\},$$

and our earlier linear-time algorithm for super-increasing knapsack problems.

Our example knapsack problem of six items is too small; typically one would have at least 250 items. The values of N and M should also be around 400 bits. However, even with parameters as large as these, the above Merkle–Hellman encryption scheme can be broken using lattice-based techniques: Suppose we wish to solve the knapsack problem given by the weights $\{w_1, \ldots, w_n\}$ and the target S. Consider the lattice L of dimension $n + 1$ generated by columns of the following matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \ldots & 0 & \frac{1}{2} \\ 0 & 0 & 1 & \ldots & 0 & \frac{1}{2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & \frac{1}{2} \\ w_1 & w_2 & w_3 & \ldots & w_n & S \end{pmatrix}.$$

Now, since we are assuming there is a solution to our knapsack problem, given by the bit vector $(b_1, \ldots, b_n)$, we know that the vector $\mathbf{y} = A \cdot \mathbf{x}$ is in our lattice, where $\mathbf{x} = (b_1, \ldots, b_n, -1)$. But the components of $\mathbf{y}$ are given by

$$y_i = \begin{cases} b_i - \frac{1}{2} & 1 \leq i \leq n \\ 0 & i = n + 1. \end{cases}$$

Hence, the vector $\mathbf{y}$ is very short, since it has length bounded by

$$\|\mathbf{y}\| = \sqrt{{y_1}^2 + \cdots + {y_{n+1}}^2} < \frac{\sqrt{n}}{2}.$$

If $\{w_1, \ldots, w_n\}$ is a set of knapsack weights then we define the density of the knapsack to be

$$d = \frac{n}{\max\{\log_2(w_i) : 1 \leq i \leq n\}}.$$

One can show that a knapsack with low density will be easy to solve using lattice basis reduction. A low-density knapsack will usually result in a lattice with relatively large discriminant, hence the vector $\mathbf{y}$ is exceptionally short in the lattice. If we now apply the LLL algorithm to the matrix A we obtain a new basis matrix A'. The first basis vector $\mathbf{a}'_1$ of this LLL-reduced basis is then likely to be the smallest vector in the lattice and so we are likely to have $\mathbf{a}'_1 = \mathbf{y}$. But given $\mathbf{y}$ we can then solve for $\mathbf{x}$ and recover the solution to the original knapsack problem. This allows us to break the Merkle–Hellman scheme, as the Merkle–Hellman construction will always produce a low-density public knapsack.

Example: As an example we take our earlier knapsack problem of

$$b_1 \cdot 62 + b_2 \cdot 93 + b_3 \cdot 81 + b_4 \cdot 88 + b_5 \cdot 102 + b_6 \cdot 37 = 174.$$

We form the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & \frac{1}{2} \\ 62 & 93 & 81 & 88 & 102 & 37 & 174 \end{pmatrix}.$$

We apply the LLL algorithm to this matrix so as to obtain the new lattice basis,

$$A' = \frac{1}{2}\begin{pmatrix} 1 & -1 & -2 & 2 & 3 & 2 & 0 \\ -1 & -3 & 0 & -2 & -1 & -2 & 0 \\ -1 & -1 & -2 & 2 & -1 & 2 & 0 \\ 1 & -1 & -2 & 0 & -1 & -2 & -2 \\ 1 & -1 & 0 & 2 & -3 & 0 & 4 \\ 1 & 1 & 0 & -2 & 1 & 2 & 0 \\ 0 & 0 & -2 & 0 & 0 & -2 & 2 \end{pmatrix}.$$

We write

$$y = \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix},$$

and compute

$$x = A^{-1} \cdot y = \begin{pmatrix} 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

So we see that we can take $(b_1, b_2, b_3, b_4, b_5, b_6) = (0, 1, 1, 0, 0, 0)$, as a solution to our knapsack problem.

17.3. Worst-Case to Average-Case Reductions

If we are going to get any further using complexity-theoretic hard problems we are going to need a way of ensuring that we are choosing instances which are from the set of worst cases. Just as we did for the RSA and DDH problems above, the way we do this is to provide a reduction from a worst-case problem to an average-case problem. However, unlike the above two number-theoretic cases our reduction will not be to the same problem, but from an instance of a worst-case problem of type X to an average-case instance of a problem of type Y.

The goal is to show that problem Y is *always* hard, no matter how we set it up. So we want to show that Y is hard *on average*. We thus take a problem X which we believe to be hard in the worst case, and then we show how to solve any such instance (in particular a worst-case instance) using an oracle which solves an average case instance of problem Y. The key insight is that if we do not think an algorithm for the worst case of problem X can exist, then this leads us to conclude there cannot be an algorithm to solve the average case of problem Y.

We will only vaguely sketch the ideas related to these concepts in this section, as the technical details are rather involved. Let us concentrate on a lattice-based problem, and let B denote the input public basis matrix of some hard problem (for example we want to solve the shortest vector problem in the lattice generated by the columns of B). Now B may not generate a "random lattice", in fact it could be a lattice for which the complexity of solving the SVP problem is at its worst. From B we want to construct another lattice problem which "looks average".

The key idea is that we want to generate a lattice basis A, for a new lattice, such that the new lattice basis is "related" to the basis B (so we can map solutions to a problem in the lattice generated by A into a solution to a problem in the lattice generated by B), but that the lattice generated by A is "uniformly random" in some sense. To generate a random lattice we just need to pick vectors at random, and take the lattice generated by the vectors. So we need some process to generate random vectors, which are not "too" random, but are random enough to fool an algorithm which solves average-case problems.

Consider the lattice $\mathcal{L}(B)$ and pick a random vector $\mathbf{x}$ in the lattice; do not worry how this is done since $\mathcal{L}(B)$ has infinitely many vectors. Now pick a non-lattice vector close to $\mathbf{x}$, in particular pick an error vector $\mathbf{e}$ chosen with a Gaussian distribution, with mean zero and some "small" standard deviation. See Figure 17.1 for an example of the resulting probability distribution for a two-dimensional lattice[2]. Now make the standard deviation get larger and larger, and eventually we will get a distribution which "looks" uniform but which is related to the original lattice basis B; see Figure 17.2.

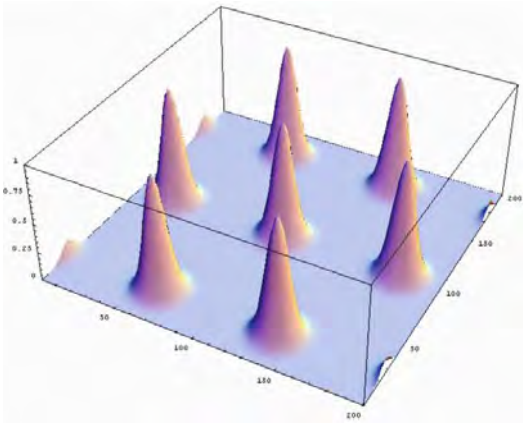


FIGURE 17.1. Perturbing a lattice by a Gaussian

We shall use this technique to give a flavour of how one can use an oracle to solve the small integer solution problem in the *average case*, so as to solve *any* approximate shortest vector problem, i.e. possibly in the *worst case*. At this point you may want to go back to the discussion around Definitions 5.4 and 5.8 to recap some basic facts.

[2]I thank Oded Regev and Noah Stephens-Davidowitz for the use of their pictures here.

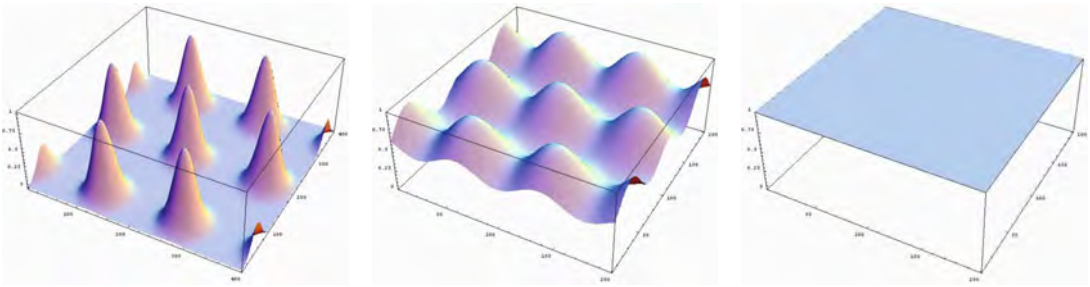


FIGURE 17.2. Increasing the Gaussian's standard deviation

Recall that for a matrix $A \in (\mathbb{Z}/q\mathbb{Z})^{n\times m}$ the SIS problem is to find a short vector $\mathbf{z} \in \mathbb{Z}^m$ such that $A \cdot \mathbf{z} = \mathbf{0} \pmod q$, where we shall assume that q is a prime. Let β denote the bound on the norm of $\mathbf{z}$ for the specific SIS instance. We will let $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ denote our oracle which solves SIS in the average case.

We remarked, in Chapter 5, that solving SIS was equivalent to finding an equivalently short vector in the lattice

$$\Lambda_q^{\perp}(A) = \{\mathbf{z} \in \mathbb{Z}^m : A \cdot \mathbf{z} = \mathbf{0} \pmod q\}.$$

The lattice $\Lambda_q^{\perp}(A)$ has determinant q^n, and hence if we select $\beta = \sqrt{m} \cdot q^{n/m}$ then the SIS oracle $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ is guaranteed to output a solution. We want to use this oracle to solve *any* approximate shortest vector problem. In particular this means that if we can find short vectors in the lattice $\Lambda_q^{\perp}(A)$ for a matrix A chosen *uniformly at random* from $(\mathbb{Z}/q\mathbb{Z})^{n\times m}$ then we can find short vectors in *any* lattice. To do this we will go via another problem, related to the shortest vector problem, namely the *shortest independent vectors problem*.

Definition 17.5 (Shortest Independent Vectors Problem (SIVP))**.** *Given a lattice L and an integer γ the SIVP problem SIVP_γ is to find a basis B of a full-rank sub-lattice such that the length of all vectors in B is less than $\gamma \cdot \lambda_n(L)$, where $\lambda_n(L)$ is the nth successive minimum of the lattice.*

Recall the transference theorem of Banaszczyk from Chapter 5: this stated that for all n-dimensional lattices we have

$$1 \leq \lambda_1(L) \cdot \lambda_n(L^*) \leq n.$$

Hence, if we can approximate the nth successive minimum of a lattice, which is what a solution to the SIVP_γ problem does, then we can approximate the shortest vector problem of its dual, and vice versa.

So we now restrict ourselves to considering the SIVP_γ problem, which we will solve for $\gamma = 16 \cdot n \cdot \sqrt{m} \cdot c_2$, for a constant c_2 defined below. We let $B \in \mathbb{Z}^{n\times n}$ denote the input matrix to our SIVP_γ problem, consisting of the n column vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. We let $\|B\| = \max_i \|\mathbf{b}_i\|$ and our goal is to find a lattice basis with $\|B\| \leq \gamma \cdot \lambda_n(L)$. We will do this by successively replacing B by a basis B' with $\|B'\| \leq \|B\|/2$. This is done by finding a vector $\mathbf{y}$ in the lattice of size bounded by $\|B\|/2$. Then, if the output vector $\mathbf{y}$ is not independent of the largest vector in B, we can replace the largest vector in B with the new vector. Thus we obtain a basis with a smaller value of $\|B\|$ and we can repeat. It turns out that the probability that the output vector is not independent of the largest vector in B is very small.

Our method proceeds as follows:

- Pick $m \leftarrow c_1 \cdot n \cdot \log q$, for some constant $0 < c_1 < 1$. We will not worry about the precise details/constants in this overview. We are just giving the basic ideas. With this choice of m we have that $q^{n/m} = \exp(1/c_1) = c_2$.
- $\beta \leftarrow \sqrt{m} \cdot q^{n/m} = \sqrt{m} \cdot c_2$.
- Pick $q \approx 4 \cdot n^{1.5} \cdot \beta$.

- For $1 \leq i \leq m$ perform the following steps:
  - Pick $\mathbf{e}_i$ for $1 \leq i \leq m$ to be n-dimensional vectors with coefficients picked from an n-dimensional Gaussian distribution with standard deviation $\sigma = \sqrt{n} \cdot \|B\|/(2 \cdot q)$.
  - Set $\mathbf{v}_i \leftarrow (\mathbf{e}_i \pmod B) - \mathbf{e}_i$, where $\mathbf{e}_i \pmod B$ means reduce $\mathbf{e}_i$ into the fundamental parallelepiped defined by the basis B. Note that $\mathbf{v}_i \in L(B)$.
  - Now set $\mathbf{u}_i \leftarrow (\mathbf{e}_i \pmod B) - \frac{1}{q} \cdot B \cdot \mathbf{a}_i$ for some integer vector $\mathbf{a}_i$, by rounding $(\mathbf{e}_i \pmod B)$ to the nearest lattice point of $L(\frac{1}{q} \cdot B)$.
- $A \leftarrow [\mathbf{a}_1, \ldots, \mathbf{a}_m]$.
- $\mathbf{z} \leftarrow \mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$.
- $\mathbf{y} \leftarrow \sum z_i \cdot (\mathbf{e}_i + \mathbf{u}_i)$.
- Output $\mathbf{y}$.

We need to show that $\mathbf{y}$ is indeed a lattice vector, that it has norm less than $\|B\|/2$, and that the SIS oracle $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ will actually work as expected. Let us look at these three points in turn:

- $\mathbf{y} \in L(B)$: Recall that the vector $\mathbf{z}$ output by the $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ oracle will satisfy $A \cdot \mathbf{z} = \mathbf{0} \pmod q$. Thus there is a $\mathbf{w} \in \mathbb{Z}^n$ with $A \cdot \mathbf{z} = q \cdot \mathbf{w}$.

  $$B \cdot \mathbf{w} = \frac{1}{q} \cdot B \cdot (q \cdot \mathbf{w}) = \frac{1}{q} \cdot B \cdot A \cdot \mathbf{z} = \frac{1}{q} \cdot B \cdot \sum \mathbf{a}_i \cdot z_i = \sum (\mathbf{e}_i + \mathbf{u}_i + \mathbf{v}_i)\, z_i.$$

  Thus, since $B \cdot \mathbf{w}$ is a lattice vector, and so is $\mathbf{v}_i$, we have that $\mathbf{y}$ must also be a lattice vector.
- $\|\mathbf{y}\| \leq \|B\|/2$: We combine two facts to achieve this bound:
  (1) First note that the tuple $(\mathbf{a}_i, \mathbf{e}_i, \mathbf{u}_i, \mathbf{v}_i)$ satisfies $\mathbf{e}_i + \mathbf{u}_i + \mathbf{v}_i = \frac{1}{q} \cdot B \cdot \mathbf{a}_i$, and that, since the maximum length of a basis vector of $L(\frac{1}{q} \cdot B)$ is bounded by $\|B\|/q$, we have $\|\mathbf{u}_i\| \leq n \cdot \|B\|/q$.
  (2) Secondly, since $\mathbf{e}_i$ has been chosen from an n-dimensional Gaussian with standard deviation σ we have that, with very high probability[3], $\|\mathbf{e}_i\| \leq 2 \cdot \sqrt{\sigma} \cdot \sqrt{n}$.

  Combining these two facts together, and using the fact that $\sigma = \sqrt{n} \cdot \|B\|/(2 \cdot q)$, we find that $\|\mathbf{e}_i + \mathbf{u}_i\| \leq 2 \cdot n \cdot \|B\|/q$. Then we use that the output of $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ satisfies $\|\mathbf{z}\| \leq \beta$, to show

  $$\begin{aligned} \|\sum (\mathbf{e}_i + \mathbf{u}_i) \cdot z_i\| &\leq \sum \|\mathbf{e}_i + \mathbf{u}_i\| \cdot |z_i| \leq \sqrt{n} \cdot \|\mathbf{z}\| \cdot \frac{2 \cdot n \cdot \|B\|}{q} \\ &\leq n^{1.5} \cdot \beta \cdot \frac{2 \cdot \|B\|}{q} \leq \frac{\|B\|}{2}. \end{aligned}$$

  The last inequality follows by our choice of β.
- The SIS oracle $\mathcal{O}_{\mathsf{SIS}}(A, q, \beta)$ is "well behaved": By choice of β there is a solution to the SIS problem for the input matrix A, so we only require that the oracle finds a solution. The SIS oracle will be well behaved, by assumption, if the input is a random SIS instance, i.e. the matrix A looks like it was selected at random from the set of all matrices modulo q. This is where the complicated analysis comes in. The basic idea is that since the standard deviation σ is chosen to be relatively large compared to the lattice basis B, the statistical distance between the vectors $\mathbf{e}_i$ modulo $L(B)$ and the uniform distribution of $\mathbb{R}^n$ modulo $L(B)$ is very small. Then since $\mathbf{a}_i$ is simply the rounding of these values to a grid point, they too act like uniform random vectors. We also need to show that the new vector is linearly independent of the other vectors, again something which we skip over here.

  So how large must σ be? The analysis says that all will be fine with the SIS oracle if we take $\sigma = \sqrt{n} \cdot \|B\|/(2 \cdot q) \geq 2 \cdot \lambda_n(L)$. But this will imply that our input matrix B

[3] In fact exponentially high probability, the estimate being obtained from examining the tail of the Gaussian distribution.

must satisfy $\|B\| > 4 \cdot q \cdot \lambda_n(L)/\sqrt{n} = \gamma \cdot \lambda_n(L)$. Thus our reduction will work as long as the basis satisfies this condition. Hence the reduction will stop as soon as we have $\|B\| \leq \gamma \cdot \lambda_n(L)$, i.e. as soon as we have solved the input SIVP_γ problem.

17.4. Learning With Errors (LWE)

Another lattice-based problem with a similar worst-case to average-case connection is the Learning With Errors (LWE) problem. An oracle to solve the average-case complexity of this problem can be used to solve the Bounded-Distance Decoding (BDD) problem in a related lattice, although the reduction is far more involved than even our sketch of the SIS to SVP reduction above.

To define the LWE problem we first need to define an error distribution, which we shall denote by $D_{\mathbb{Z}^n,\sigma}$; this distribution produces n-dimensional integer vectors whose coefficients are distributed approximately like a normal distribution with mean zero and standard deviation σ. One should think of the elements output with high probability by the distribution $D_{\mathbb{Z}^n,\sigma}$ as having "small" coefficients. We have two LWE problems: the search and decision problems; we can define advantages for adversaries against these problems in the usual way.

Definition 17.6 (LWE Search Problem)**.** *Pick* $\mathbf{s} \leftarrow (\mathbb{Z}/q\mathbb{Z})^m, \mathbf{e} \leftarrow D_{\mathbb{Z}^n,\sigma}$ *and* $A \leftarrow (\mathbb{Z}/q\mathbb{Z})^{n\times m}$*, where* $n > m$*, and set* $\mathbf{b} \leftarrow A \cdot \mathbf{s} + \mathbf{e} \pmod q$*. The search problem is given the pair* $(A, \mathbf{b})$ *to output the vector* $\mathbf{s}$*.*

Definition 17.7 (LWE Decision Problem)**.** *Given* $(A, \mathbf{b})$ *where* $A \in (\mathbb{Z}/q\mathbb{Z})^{n\times m}$*, where* $n > m$ *and* $\mathbf{b} \in (\mathbb{Z}/q\mathbb{Z})^n$*; determine which of the following two cases holds:*

(1) $\mathbf{b}$ *is chosen uniformly at random from* $(\mathbb{Z}/q\mathbb{Z})^n$*.*
(2) $\mathbf{b} \leftarrow A \cdot \mathbf{s} + \mathbf{e} \pmod q$ *where* $\mathbf{e} \leftarrow D_{\mathbb{Z}^n,\sigma}$ *and* $\mathbf{s} \leftarrow (\mathbb{Z}/q\mathbb{Z})^m$*.*

Let us first consider the search problem. Suppose the error vector $\mathbf{e}$ was zero, we would then be solving the linear system of equations $A \cdot \mathbf{s} = \mathbf{b} \pmod q$, for a matrix with more rows than columns. This will have a solution if the set of equations is consistent, which it will be by our construction. Finding that solution just requires the inversion of the matrix, and is hence an easy computational problem. So simply adding a small error vector $\mathbf{e}$ seems to make the problem very much harder. In some sense we are decoding a random linear code over $\mathbb{Z}/q\mathbb{Z}$, where the error vector is $\mathbf{e}$.

We can relate the search problem to our q-ary lattices from Chapter 5, where we let $L = \Lambda_q(A^\mathsf{T})$. We first note that recovering $\mathbf{e}$ is equivalent to recovering $\mathbf{s}$. The LWE problem then is to find a vector $\mathbf{x} \in L$ such that $\mathbf{b} - \mathbf{x} = \mathbf{e}$. So to turn this into a Bounded-Distance Decoding problem we need to bound the size of the error vector $\mathbf{e}$. However, $\mathbf{e}$ was chosen from a distribution which was a discrete approximation to the Gaussian distribution, so we can prove that its norm is not too large. In fact one can show that with very high probability $\|\mathbf{e}\| \leq 2 \cdot \sigma \cdot \sqrt{m}/\sqrt{2 \cdot \pi}$. Thus we wish to solve a BDD_α problem for $\alpha = 2 \cdot \sigma \cdot \sqrt{m}/(\sqrt{2 \cdot \pi} \cdot \lambda_1(L))$.

Before proceeding we state some basic facts, without proof, about the LWE problem.

- The decision and search problems are equivalent, i.e. given an oracle to solve the decision problem we can use it to solve the search problem.
- The secret vector $\mathbf{s}$ in the definition can be selected to come from the error distribution $D_{\mathbb{Z}^m,\sigma}$ with no reduction in hardness.
- Whilst the worst-case/average-case hardness result requires the error distribution to come from $D_{\mathbb{Z}^n,\sigma}$, in practice we can select any distribution which outputs sufficiently "small" and high-entropy vectors.

17.4.1. Public Key System Based on LWE: We can now define a public key cryptosystem whose hardness is closely related to the worst case complexity of the Bounded-Distance Decoding problem, and for which we currently do not know of any attacks which can be mounted with a quantum computer. The scheme is parametrized by integer values n, m and q, with $m \approx n \cdot \log q$ and $q > 2$ prime. In what follows for the rest of this chapter when we reduce something modulo q

we take a representative in the range $(-q/2, \ldots, q/2]$, i.e. we take the set of representatives centred around zero.

Key Generation: For the private key we select $\mathbf{s} \leftarrow (\mathbb{Z}/q\mathbb{Z})^n$. To create the public key we generate m vectors $\mathbf{a}_i \leftarrow (\mathbb{Z}/q\mathbb{Z})^n$, and m error values $e_i \leftarrow D_{\mathbb{Z}^1,\sigma}$. We set $b_i \leftarrow \mathbf{a}_i \cdot \mathbf{s} - 2 \cdot e_i \pmod q$ and output the public key $\mathfrak{pk} \leftarrow ((\mathbf{a}_1, b_1), \ldots, (\mathbf{a}_m, b_m))$.

Encryption: To encrypt a message (which will be a single bit $m \in \{0, 1\}$), the encryptor picks a subset of indicator bits in $\{1, \ldots, m\}$, i.e. he picks $\iota_i \leftarrow \{0, 1\}$ for $i = 1, \ldots, m$. The ciphertext is then the pair of values (consisting of a vector $\mathbf{c}$ and an integer d)

$$\mathbf{c} \leftarrow \sum_{i=1}^{m} \iota_i \cdot \mathbf{a}_i \quad \text{and} \quad d \leftarrow m - \sum_{i=1}^{m} \iota_i \cdot b_i.$$

Decryption: Decryption of $(\mathbf{c}, d)$ is performed by evaluating

$$\begin{aligned}
&\Big(\mathbf{c} \cdot \mathbf{s} + d \pmod q\Big) \pmod 2 \\
&\quad = \left(\left(\sum_{i=1}^{m} \iota_i \cdot \mathbf{a}_i \cdot \mathbf{s} - \iota_i \cdot b_i\right) + m \pmod q\right) \pmod 2, \\
&\quad = \left(\left(\sum_{i=1}^{m} 2 \cdot \iota_i \cdot e_i\right) + m \pmod q\right) \pmod 2, && \text{by definition of } b_i \\
&\quad = (2 \cdot \text{“small”} + m) \pmod 2 && \text{since the } e_i \text{ are small} \\
&\quad = m.
\end{aligned}$$

Note that the sum $\eta = \sum_{i=1}^{m} 2 \cdot \iota_i \cdot e_i$ will be “small” modulo q, and will not “wrap around” (i.e. exceed $q/2$ in absolute value) since the e_i have been selected suitably small. We shall call the value η the “noise” associated to the ciphertext. The reason decryption works is that this noise is small relative to the modulus q, and hence when we are taking the inner modulo q operation we obtain the value of $\eta + m$ *over the integers*. The reason for taking reduction centred around zero is to ensure this last fact. Since η is divisible by two we can recover the message m by reducing the result modulo two.

The general design of the scheme is that the public key contains a large number of encryptions of zero. To encrypt a message m we take a subset of these encryptions of zero and add them to the message, thus obtaining an encryption of m, assuming the “noise” does not get too large. It can be shown that the above scheme is IND-CPA assuming the LWE decision problem is hard.

17.4.2. Ring-LWE: The problem with the above LWE-based system is that we can only encrypt messages in $\{0, 1\}$, and that each ciphertext consists of a vector $\mathbf{c} \in (\mathbb{Z}/q\mathbb{Z})^n$ and a value $d \in \mathbb{Z}/q\mathbb{Z}$. The public key is also very large consisting of $2 \cdot m$ vectors of length n and $2 \cdot m$ integers (all of values in $\mathbb{Z}/q\mathbb{Z}$). Thus the overhead is relatively large. To reduce this overhead, and to increase the available message space, it is common to use a variant of LWE based on rings of polynomials, called *Ring-LWE*. Ring-LWE based constructions are also more efficient from a computational perspective.

Recall that normal LWE is about matrix-vector multiplications, $\mathbf{x} \longrightarrow A \cdot \mathbf{x}$. In Ring-LWE we take “special” matrices A, which arise from polynomial rings. Consider the ring of polynomials $R = \mathbb{Z}[X]/F(X)$, where $F(X)$ is an integer polynomial of degree n. The ring R is the set of polynomials of degree less than n with integer coefficients, addition being standard polynomial addition and multiplication being polynomial multiplication followed by reduction modulo $F(X)$. As a shorthand we will use R_q to denote the ring R reduced modulo q, i.e. the ring $(\mathbb{Z}/q\mathbb{Z})[X]/F(X)$.

Now there is a link between polynomial rings and matrix rings, since we can think of a polynomial as defining a vector and a matrix. In particular suppose we have two polynomials $a(X)$

and $b(X)$ in the ring R, whose product in R is $c(X)$. Clearly we can think of $b(X)$ as a vector $\mathbf{b}$, where we take the coefficients of $b(X)$ as the vector of coefficients. Similarly for $c(X)$ and a vector $\mathbf{c}$. Now the polynomial $a(X)$ can be represented by an $n \times n$ matrix, which we shall call M_a, such that $\mathbf{c} = M_a \cdot \mathbf{b}$. The mapping from R to $n \times n$ matrices which sends $a(X)$ to M_a is what mathematicians call the *matrix representation* of the ring R.

With this interpretation of polynomials as vectors and matrices we can try to translate the LWE problem above into the polynomial ring setting. We first define the *error distribution* $D_{R,\sigma}$, which now outputs polynomials of degree less than n whose coefficients are distributed approximately like a normal distribution with mean zero and standard deviation σ. Again the outputs are polynomials that with high probability have "small" coefficients. We let B_0 denote a bound on the absolute value of the coefficients sampled by the distribution $D_{R,\sigma}$; actually this is an expected bound since the distribution could sample a polynomial with huge coefficients, but this is highly unlikely. Given this distribution we can define two Ring-LWE problems, again one search and one decision.

Definition 17.8 (Ring-LWE Search Problem)**.** *Pick $a, s \in R_q$ and $e \leftarrow D_{R,\sigma}$ and set $b \leftarrow a \cdot s + e \pmod q$. The search problem is given the pair (a, b) to output the value s.*

Definition 17.9 (Ring-LWE Decision Problem)**.** *Given (a, b) where $a, b \in R_q$ determine which of the following two cases holds:*

(1) *b is chosen uniformly at random from R_q.*
(2) *$b \leftarrow a \cdot s + e \pmod q$ where $e \leftarrow D_{R,\sigma}$ and $s \leftarrow R_q$.*

We note that the value s can be selected from the distribution $D_{R,\sigma}$, and in the public key scheme we present next we will indeed do this. The reason for this modification will become apparent later.

17.4.3. Public Key System Based on Ring-LWE: We can immediately map the public key LWE-based system over to the Ring-LWE setting. However, we would like to do this in a way which also reduces the size of the public key. Recall that in the LWE encryption scheme, the public key contained a large number of encryptions of zero. What we require is a mechanism to come up with these encryptions of zero "on the fly" without needing to place them in the public key.

Key Generation: We pick coprime integers p and q with $p \ll q$, and a ring R as above, plus a standard deviation σ. The "security" will depend on the dimension n of the ring R and the size of σ and q. We do not go into these details in this book as the discussion gets rather complicated, rather quickly. The set $\{p, q, R, \sigma\}$ can be considered the domain parameters, similar to the domain parameters in ElGamal encryption. To generate an individual's public key and the private key we execute:

- $s, e \leftarrow D_{R,\sigma}$.
- $a \leftarrow R_q$.
- $b \leftarrow a \cdot s + p \cdot e \pmod q$.
- $\mathfrak{pk} \leftarrow (a, b)$, $\mathfrak{sk} \leftarrow s$.

With this scheme we will be able to encrypt arbitrary elements of the ring R_p, and we can think of the public key as a random encryption of the element zero in this ring.

Encryption: To encrypt a message $m \in R_p$ we execute

- $e_0, e_1, e_2 \leftarrow D_{R,\sigma}$.
- $c_0 \leftarrow b \cdot e_0 + p \cdot e_1 + m$.
- $c_1 \leftarrow a \cdot e_0 + p \cdot e_2$.

The ciphertext is then (c_0, c_1).

Decryption: To decrypt we compute

$$\begin{aligned}
&\Big(c_0 - c_1 \cdot s \pmod q\Big) \pmod p \\
&\quad = \Big((b \cdot e_0 + p \cdot e_1 + m) - (a \cdot e_0 + p \cdot e_2) \cdot s \pmod q\Big) \pmod p \\
&\quad = \Big(a \cdot s \cdot e_0 + p \cdot e \cdot e_0 + p \cdot e_1 + m \\
&\qquad\quad - a \cdot e_0 \cdot s - p \cdot e_2 \cdot s \pmod q\Big) \pmod p \\
&\quad = \Big(p \cdot (e \cdot e_0 + e_1 - e_2 \cdot s) + m \pmod q\Big) \pmod p \\
&\quad = \Big(p \cdot \text{“small”} + m \pmod q\Big) \pmod p \qquad \text{since } s, e, e_i \text{ are small} \\
&\quad = \Big(p \cdot \text{“small”} + m\Big) \pmod p \\
&\quad = m.
\end{aligned}$$

Another way to think of decryption is to take the vector product between the vector (c_0, c_1) and the vector $(1, -s)$, a viewpoint which we shall use below.

Notice how the “noise” associated with the ciphertext is given by

$$\eta = e \cdot e_0 + e_1 - e_2 \cdot s$$

and that this is small since we picked not only e, e_0, e_1 and e_2 from the error distribution, but also s. Also notice how we use the randomization by e_0, e_1 and e_2 to produce a random “encryption” of zero in the encryption algorithm. Thus we have not only managed to improve the message capacity from $\{0, 1\}$ to R_p, but we have also reduced the public key size as well. It can be shown that the above scheme is IND-CPA assuming the Ring-LWE decision problem is hard. We cannot make the scheme as it stands IND-CCA secure as it suffers from an issue of malleability. Indeed the malleability of the scheme is very interesting as it allows us to construct a so-called Fully Homomorphic Encryption (FHE) scheme.

17.4.4. Fully Homomorphic Encryption: Usually a scheme being malleable is a bad thing, however in some situations it is actually useful. For example, in Chapter 21 we shall see how an additively homomorphic encryption scheme can be used to implement a secure electronic voting protocol. We have seen a number of schemes so far which are multiplicatively homomorphic, e.g. naive RSA encryption, ElGamal encryption and one which is additively homomorphic, namely Paillier encryption. A scheme which is *both* additively and multiplicatively homomorphic is called *fully homomorphic*.

More formally let R be a ring with operations $(+, \cdot)$, and let E and D denote the public key encryption and decryption operations, which encrypt and decrypt elements of R. So we have

$$D\Big(E(m, \mathfrak{pk}),\ \mathfrak{sk} \Big) = m.$$

A scheme is said to be fully homomorphic if there are two functions Add and Mult which each take as input two ciphertexts, and respectively return ciphertexts as output such that for all $m_1, m_2 \in R$ we have

$$\begin{aligned}
D\Big(\mathsf{Add}\Big(E(m_1, \mathfrak{pk}),\ E(m_2, \mathfrak{pk})\Big),\ \mathfrak{sk} \Big) &= m_1 + m_2, \\
D\Big(\mathsf{Mult}\Big(E(m_1, \mathfrak{pk}),\ E(m_2, \mathfrak{pk})\Big),\ \mathfrak{sk} \Big) &= m_1 \cdot m_2.
\end{aligned}$$

To see why a fully homomorphic encryption scheme might be useful, consider the following situation. Suppose Alice encrypts some information m and stores the ciphertext c on a remote server. Then later on Alice wants to compute some function F on the message, for example she may want to

know whether the message was an email from Bob saying "I love you". It would appear that Alice needs to retrieve the ciphertext from the server, decrypt it, and then perform the calculation. This might not be convenient (especially in the case when the message is a huge database, say).

With a fully homomorphic encryption scheme Alice can instead send the function F to the server, and the server can then compute a new ciphertext c_F which is the encryption of $F(m)$. Amazingly this can be done *without* the server ever needing to obtain the message m. The reason for this is that *every* function can be expressed as a series of additions and multiplications over a ring, and hence we can express F as such a series. Then we apply the addition and multiplication of the fully homomorphic encryption scheme to obtain a ciphertext which decrypts to $F(m)$.

The idea of constructing a fully homomorphic encryption scheme dates back to the early days of public key cryptography. But it was not until 2009 that even a theoretical construction was proposed, by Craig Gentry. Gentry's original construction was highly complicated, but now we can present relatively simple (although not very efficient) schemes based on the LWE and Ring-LWE problems.

Let us look again at our Ring-LWE encryption scheme. All we need look at is what a ciphertext is; we can ignore the encryption procedure. A ciphertext is a pair $c = (c_0, c_1)$ such that

$$c_0 - c_1 \cdot s = m + p \cdot \eta_c \pmod{q},$$

where η_c is the noise associated with the ciphertext c. It is clear that this encryption scheme is additively homomorphic to some extent, since we have for two ciphertexts $c = (c_0, c_1)$ and $c' = (c'_0, c'_1)$ encrypting m and m' that

$$\begin{aligned}\left(c_0 + c'_0\right) - \left(c_1 + c'_1\right) \cdot s &= \left(c_0 - c_1 \cdot s\right) + \left(c'_0 - c'_1 \cdot s\right) \\ &= \left(m + p \cdot \eta_c\right) + \left(m' + p \cdot \eta_{c'}\right) \pmod{q} \\ &= \left(m + m'\right) + p \cdot \left(\eta_c + \eta_{c'}\right) \pmod{q}.\end{aligned}$$

In particular we can keep adding ciphertexts together componentwise and they will decrypt to the sum of the associated messages, up until the point when the sum of the noises becomes too large, when decryption will fail. We call a system for which we can perform a limited number of such additions a *somewhat* additively homomorphic scheme.

It is not immediately obvious that the above scheme will support multiplication of ciphertexts. To do so we need to augment the public key a little. As well as the pair (a, b) we also include in the public key an encryption of $p^i \cdot s^2$, for $i \in [0, \ldots, \lceil \log_p q \rceil]$, in the following way. We pick $a'_i \leftarrow R_q$ and $e'_i \leftarrow D_{R,\sigma}$ and set $b'_i \leftarrow a'_i \cdot s + p \cdot e'_i + p^i \cdot s_i{}^2 \pmod{q}$. Note that these are not really encryptions of $p^i \cdot s^2$, since $p^i \cdot s^2 \pmod{q}$ is not even an element of the plaintext space R_p. Despite this problem the following method will work, and it is useful to think of (a'_i, b'_i) as an encryption of $p^i \cdot s^2$.

Now suppose we have two ciphertexts $c = (c_0, c_1)$ and $c' = (c'_0, c'_1)$ encrypting m and m'. We first form the tensor product ciphertext $c \otimes c' = (c_0 \cdot c'_0,\ c_0 \cdot c'_1,\ c_1 \cdot c'_0,\ c_1 \cdot c'_1) = (d_0,\ d_1,\ d_2,\ d_3)$. This four-dimensional ciphertext will decrypt with respect to the "secret key" vector $(1,\ -s) \otimes (1,\ -s) = (1,\ -s,\ -s,\ s^2)$, since

$$\begin{aligned}d_0 - d_1 \cdot s - d_2 \cdot s + d_3 \cdot s^2 &= c_0 \cdot c'_0 - c_0 \cdot c'_1 \cdot s - c_1 \cdot c'_0 \cdot s + c_1 \cdot c'_1 \cdot s^2 \pmod{q} \\ &= \left(c_0 - c_1 \cdot s\right) \cdot \left(c'_0 - c'_1 \cdot s\right) \pmod{q} \\ &= \left(m + p \cdot \eta_c\right) \cdot \left(m' + p \cdot \eta_{c'}\right) \pmod{q} \\ &= m \cdot m' + p \cdot \left(m \cdot \eta_{c'} + m' \cdot \eta_c + p \cdot \eta_c \cdot \eta_{c'}\right) \pmod{q}.\end{aligned}$$

So we see that the "ciphertext" $c \otimes c'$ decrypts to the product of m and m' under the secret key $(1, -s) \otimes (1, -s)$ assuming the noise is not too large. Since $p \ll q$ we have that the noise grows roughly as $p \cdot \eta_c \cdot \eta_{c'}$. The problem is that the ciphertext is now twice as large; we would like to reduce to a ciphertext with two components which decrypts in the usual way via the secret vector $(1, -s)$.

To achieve this goal we use the data with which we augmented the public key, namely the encryption (a'_i, b'_i) of $p^i \cdot s^2$. Given the four-dimensional ciphertext (d_0, d_1, d_2, d_3) we write d_3 in its base-p representation, i.e.

$$d_3 = \sum_{i=0}^{\lceil \log_p q \rceil} p^i \cdot d_{3,i},$$

where $d_{3,i} \in R_p$. We then set

$$f_0 \leftarrow d_0 + \sum_{i=0}^{\lceil \log_p q \rceil} d_{3,i} \cdot b'_i,$$

$$f_1 \leftarrow d_1 + d_2 + \sum_{i=0}^{\lceil \log_p q \rceil} d_{3,i} \cdot a'_i.$$

Then the ciphertext $f = (f_0, f_1)$ decrypts to

$$\begin{aligned} f_0 - f_1 \cdot s &= \left(d_0 + \sum_{i=0}^{\lceil \log_p q \rceil} d_{3,i} \cdot b'_i\right) - \left(d_1 + d_2 + \sum_{i=0}^{\lceil \log_p q \rceil} d_{3,i} \cdot a'_i\right) \cdot s \pmod q \\ &= d_0 - d_1 \cdot s - d_2 \cdot s + \sum_{i=0}^{\lceil \log_p q \rceil} \left(d_{3,i} \cdot s^2 + d_{3,i} \cdot a'_i \cdot s + p \cdot e'_i \cdot d_{3,i} - d_{3,i} \cdot a'_i \cdot s\right) \pmod q \\ &= \left(d_0 - d_1 \cdot s - d_2 \cdot s + d_3 \cdot s^2\right) + p \cdot \sum_{i=0}^{\lceil \log_p q \rceil} e'_i \cdot d_{3,i}. \end{aligned}$$

So we obtain a ciphertext which decrypts to $m \cdot m'$ using the above equation, but with noise term

$$\eta_f = m \cdot \eta_{c'} + m' \cdot \eta_c + p \cdot \eta_c \cdot \eta_{c'} + \sum_{i=0}^{\lceil \log_p q \rceil} e'_i \cdot d_{3,i}.$$

We need to estimate the size of η_f as a polynomial over the integers. To do this we use the norm $\|f\|$ which returns the maximum of the absolute value of the coefficients of f. Suppose the noise is bounded by $\|\eta_c\|, \|\eta_{c'}\| \leq B$ for the two ciphertexts passed into the multiplication. Since $m, m', d_{3,i} \in R_p$ we have that $\|m\|, \|m'\|, \|d_{3,i}\| \leq p/2$. By definition we have $\|e_i\| \leq B_0$, and so we have

$$\begin{aligned} \|\eta_f\| &\leq \frac{p \cdot B}{2} + \frac{p \cdot B}{2} + p \cdot B^2 + \frac{\lceil \log_p q \rceil \cdot B_0 \cdot p}{2} \\ &= p \cdot \left(B + B^2 + \frac{\lceil \log_p q \rceil \cdot B_0}{2}\right) \end{aligned}$$

which is about $p \cdot B^2$. Thus when adding ciphertexts with noise B the noise increases to $2 \cdot B$, but when multiplying we get noise $p \cdot B^2$.

Eventually, if we keep performing operations the noise will become too large and decryption will fail, in particular when the noise is larger than $q/4$. Thus if we make q larger we can ensure that we can evaluate more operations homomorphically. However, increasing q means we also need to increase the degree of our ring to ensure the hardness of the Ring-LWE problem. In this way we

obtain a Somewhat Homomorphic Encryption (SHE) scheme, i.e. one which can evaluate a limited number of addition and multiplication operations. To obtain a Fully Homomorphic Encryption (FHE) scheme we need to be able to perform an arbitrary number of additions and multiplications.

The standard theoretical trick to obtain an FHE scheme from an SHE scheme is via a technique called bootstrapping. We outline the basic idea behind bootstrapping as the details are very complicated, and the technique is currently impractical bar for toy examples. The decryption algorithm D is a function of the ciphertext c and the secret key s. We can thus represent the function D as an arithmetic circuit C (i.e. a circuit involving only additions and multiplications) of representations of c and s. To do this we first have to convert the representations of c and s to be elements of R_p, but taking the bit representations and using a binary circuit is an easy (but inefficient) way of doing this.

Now suppose C is "simple" enough to be evaluated by the SHE scheme, and we can do a little more than just evaluate C, say perform another multiplication. Our Ring-LWE scheme indeed has a simple decryption circuit, since the only difficult part is reduction modulo q and then modulo p, the main decryption equation being linear. We can then "refresh" or "bootstrap" a ciphertext c by homomorphically evaluating the circuit C on the *actual* ciphertext c and the *encryption* of the secret key s. In other words, we add to the public key the encryption of the secret key, and then by homomorphically evaluating C we obtain a new encryption of the message encrypted by c, but with a smaller noise value.

Chapter Summary

- Complexity theory deals with the worst-case behaviour of algorithms to solve a given decision problem; some problems are easy on average, but there exist certain instances which are very hard to solve.
- Problems such as the RSA and DDH problems are hard on average, since they possess random self-reductions from a given instance of the problem to a random instance of the problem.
- Cryptographic systems based on knapsack problems have been particularly notorious from this perspective, as one can often use lattice basis reduction to break them.
- To obtain secure systems based on lattices one should select lattice problems which have worst-case to average-case reductions.
- One such problem is the LWE problem, another is the closely related Ring-LWE problem.
- These lattice problems allow us to construct fully homomorphic encryption schemes and schemes which appear to resist quantum attacks.

Further Reading

A nice introduction to complexity theory can be found in the book by Goldreich. A discussion of knapsack based systems and how to break them using lattices can be found in the survey article by Odlyzko. A good survey of the more modern constructive applications of lattices, including LWE, can be found in the paper by Micciancio and Regev.

O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

A. Odlyzko. *The rise and fall of knapsack cryptosystems*. In Cryptology and Computational Number Theory, Proc. Symposia in Applied Maths, Volume 42, AMS, 1990.

D. Micciancio and O. Regev. *Lattice based cryptography*. Post-Quantum Cryptography, 147–191, Springer, 2009.

CHAPTER 18

Certificates, Key Transport and Key Agreement

Chapter Goals

- To understand the problems associated with managing and distributing secret keys.
- To introduce the notion of digital certificates and a PKI.
- To show how an implicit certificate scheme can operate.
- To learn about key distribution techniques based on symmetric-key-based protocols.
- To introduce key transport based on public key encryption.
- To introduce Diffie–Hellman key exchange, and its various variations.
- To introduce the symbolic and computational analysis of protocols.

18.1. Introduction

We can now perform encryption and authentication using either public key techniques or symmetric key techniques. However, we have not addressed how parties actually obtain the public key of an entity, or obtain a shared symmetric key. When using hybrid ciphers in Section 16.3 we showed how a symmetric key could be transported to another party, and then that symmetric key used to encrypt a *single* message. However, we did not address what happens if we want to use the symmetric key many times, or use it for authentication etc. Nor did we address how the sender would know that the public key of the receiver he was using was genuine.

In this chapter we present methodologies to solve all of these problems. In doing so we present our first examples of what can be called "cryptographic protocols". A cryptographic protocol is an exchange of messages which achieves some cryptographic goal. Up until now we have created single shot mechanisms (encryption, MAC, signatures etc.) which have not required interaction between parties.

In dealing with cryptographic protocols we still need to define what we mean by something being secure, and when presenting a protocol we need to present a proof as to why we believe the protocol meets our security definition. However, unlike the proofs for encryption etc. that we have met before, the proofs for protocols are incredibly complex. They fall into one of two camps[1]:

- In the first camp are so-called "symbolic methods", these treat underlying primitives such as encryption as perfect black boxes and then try to show that a protocol is "secure". However, as they work at a very high level of abstraction they do not really prove security, they simply enable one to find attacks on a protocol relatively easily. This is because in this camp one shows a protocol is insecure by exhibiting an attack.
- The second camp is like our security games for encryption etc. We define a game, with an adversary. The adversary has certain powers (given by oracles), and has a certain goal. We then present proofs that the existence of such an adversary implies the existence of an algorithm which can solve a hard problem. These proofs provide a high degree of assurance

[1]There is a third camp called the *simulation paradigm*, typified by something called the Universal Composability (UC) framework. This camp is beyond the scope of this book however.

that the protocol is sound, and any security flaws will come from implementation aspects as opposed to protocol design issues.

Before we continue we need to distinguish between different types of keys. The following terminology will be used throughout this chapter and beyond:

- **Static (or Long-Term) Keys:** These are keys which are to be in use for a long time period. The exact definition of long will depend on the application, but this could mean from a few hours to a few years. The compromising of a static key is usually considered to be a major problem, with potentially catastrophic consequences.
- **Ephemeral, or Session (or Short-Term) Keys:** These are keys which have a short lifetime, maybe a few seconds or a day. They are usually used to provide confidentiality for a given time period. The compromising of a session key should only result in the compromising of that session's secrecy and it should not affect the long-term security of the system.

Key distribution is one of the fundamental problems of cryptography. There are a number of solutions to this problem; which solution one chooses depends on the overall system.

- **Physical Distribution:** Using trusted couriers or armed guards, keys can be distributed using traditional physical means. Until the 1970s this was in effect the only secure way of distributing keys at system setup. It has a large number of physical problems associated with it, especially scalability, but the main drawback is that security no longer rests with the key but with the courier. If we can bribe, kidnap or kill the courier then we have broken the system.
- **Distribution Using Symmetric Key Protocols:** Once some secret keys have been distributed between a number of users and a trusted central authority, we can use the trusted authority to help generate keys for any pair of users as the need arises. Protocols to perform this task will be discussed in this chapter. They are usually very efficient but have some drawbacks. In particular they usually assume that both the trusted authority and the two users who wish to agree on a key are both online. They also still require a physical means to set up the initial keys.
- **Distribution Using Public Key Protocols:** Using public key cryptography, two parties, who have never met or who do not trust any one single authority, can produce a shared secret key. This can be done in an online manner, using a key exchange protocol. Indeed this is the most common application of public key techniques for encryption. Rather than encrypting large amounts of data by public key techniques we agree a key by public key techniques and then use a symmetric cipher to actually do the encryption, a methodology we saw earlier when we discussed hybrid encryption.

To understand the scale of the problem, if our system is to cope with n separate users, and each user may want to communicate securely with any other user, then we require

$$\frac{n \cdot (n-1)}{2}$$

separate symmetric keys. This soon produces huge key management problems; a small university with around 10 000 students would need to have around fifty million separate secret keys.

With a large number of keys in existence one finds a large number of problems. For example what happens when your key is compromised? In other words, if someone else has found your key. What can you do about it? What can they do? Hence, a large number of keys produces a large key management problem.

One solution is for each user to hold only one key with which they communicate with a central authority, hence a system with n users will only require n keys. When two users wish to communicate, they generate a secret key which is only to be used for that message, a so-called session key

or ephemeral key. This session key can be generated with the help of the central authority using one of the protocols that appear later in this chapter.

As we have mentioned already the main problem is one of managing the secure distribution of keys. Even a system which uses a trusted central authority needs some way of getting the keys shared between the centre and each user out to the user. One possible solution is key splitting (more formally called *secret sharing*) where we divide the key into a number of shares

$$k = k_1 \oplus k_2 \oplus \cdots \oplus k_r.$$

The shares are then distributed via separate routes. The beauty of this is that an attacker needs to attack all the routes so as to obtain the key. On the other hand attacking one route will stop the legitimate user from recovering the key. We will discuss secret sharing in more detail in Chapter 19.

One issue one needs to consider when generating and storing keys is the key lifetime. A general rule is that the longer the key is in use the more vulnerable it will be and the more valuable it will be to an attacker. We have already touched on this when mentioning the use of session keys. However, it is important to destroy keys properly after use. Relying on an operating system to delete a file by typing `del` or `rm` does not mean that an attacker cannot recover the file contents by examining the hard disk. Usually deleting a file does not destroy the file contents, it only signals that the file's location is now available for overwriting with new data. A similar problem occurs when deleting memory in an application.

This (rather lengthy) chapter is organized in the following sections. We first introduce the notion of certificates and a Public Key Infrastructure (PKI); such techniques allow parties to obtain authentic public keys. We then discuss a number of protocols which allow parties to agree new symmetric ephemeral keys, given already deployed static symmetric keys. Then we discuss how to obtain new symmetric ephemeral keys, given existing public keys (which can be authenticated via a PKI). Then to give a flavour of how protocols are analysed we present a simple symbolic method called BAN Logic and apply it to one of our symmetric-key-based protocols. We then give a detailed exposition of how one uses game style security definitions to show that the public-key-based protocols are secure. Note that we can use symbolic methods to analyse public-key-based techniques, and game style security to analyse symmetric-key-based techniques. However, for reasons of space we only give a limited set of examples.

18.2. Certificates and Certificate Authorities

When using a symmetric key system we assume we do not have to worry about which key belongs to which party. It is tacitly assumed that if Alice holds a long-term secret key K_{ab} which she thinks is shared with Bob, then Bob really does have a copy of the same key. This assurance is often achieved using a trusted physical means of long-term key distribution, for example using armed couriers.

In a public key system the issues are different. Alice may have a public key which she thinks is associated with Bob, but we usually do not assume that Alice is one hundred percent certain that it really belongs to Bob. This is because we do not, in the public key model, assume a physically secure key distribution system. After all, that was one of the points of public key cryptography in the first place: to make key management easier. Alice may have obtained the public key she thinks belongs to Bob from Bob's web page, but how does she know the web page has not been spoofed?

The process of linking a public key to an entity or principal, be it a person, machine or process, is called binding. One way of binding, common in many applications where the principal really does need to be present, is by using a physical token such as a smart card. Possession of the token, and knowledge of any PIN/password needed to unlock the token, is assumed to be equivalent to being the designated entity. This solution has a number of problems associated with it, since cards can

be lost or stolen, which is why we protect them using a PIN (or in more important applications by using biometrics). The major problem is that most entities are non-human; they are computers and computers do not carry cards. In addition many public key protocols are performed over networks where physical presence of the principal (if they are human) is not something one can test.

Hence, some form of binding is needed which can be used in a variety of very different applications. The main binding tool in use today is the *digital certificate*. In this a special trusted third party, or TTP, called a certificate authority, or CA, is used to vouch for the validity of the public keys. A certificate authority system works as follows:

- All users have a trusted copy of the public key of the CA. For example these come embedded in your browser when you buy your computer, and you "of course" trust the vendor of the computer and the manufacturer of the software on your computer.
- The CA's job is to digitally sign data strings containing the following information

  (Alice, Alice's public key).

  This data string and the associated signature is called a digital certificate. The CA will only sign this data if it truly believes that the public key really does belong to Alice.
- When Alice now sends you her public key, contained in a digital certificate, you now trust that the purported key really is that of Alice, since you trust the CA to do its job correctly.

This use of a digital certificate binds the name "Alice" with the key. Public key certificates will typically (although not always) be stored in repositories and accessed as required. For example, most browsers keep a list of the certificates that they have come across. The digital certificates do not need to be stored securely since they cannot be tampered with as they are digitally signed.

To see the advantage of certificates and CAs in more detail consider the following example of a world without a CA. In the following discussion we break with our colour convention for a moment and now use red to signal public keys which must be obtained in an authentic manner and blue to signal public keys which do not need to be obtained in an authentic manner.

In a world without a CA you obtain many individual public keys from each individual in some authentic fashion. For example

6A5DEF....A21	Jim Bean's public key,
7F341A....BFF	Jane Doe's public key,
B5F34A....E6D	Microsoft's update key.

Hence, each key needs to be obtained in an authentic manner, as does every new key you obtain.

Now consider the world with a CA. You obtain a single public key in an authentic manner, namely the CA's public key. We shall call our CA Ted since he is Trustworthy. You then obtain many individual public keys, signed by the CA, in possibly an unauthentic manner. For example they could be attached at the bottom of an email, or picked up whilst browsing the web.

A45EFB....C45	Ted's totally trustworthy key,
6A5DEF....A21	Ted says "This is Jim Bean's public key",
7F341A....BFF	Ted says "This is Jane Doe's public key",
B5F34A....E6D	Ted says "This is Microsoft's update key".

If you trust Ted's key and you trust Ted to do his job correctly then you trust all the public keys you hold to be authentic.

In general a digital certificate is not just a signature on the single pair (Alice, Alice's public key); one can place all sorts of other, possibly application specific, information into the certificate. For example it is usual for the certificate to contain the following information.

- User's name,
- User's public key,

- Is this an encryption or signing key?
- Name of the CA,
- Serial number of the certificate,
- Expiry date of the certificate,
- ....

Commercial certificate authorities exist that will produce a digital certificate for your public key, often after payment of a fee and some checks on whether you are who you say you are. The certificates produced by commercial CAs are often made public, so one could call them public "public key certificates", in that their use is mainly over open public networks. CAs are also used in proprietary closed systems, for example in debit/credit card systems or by large corporations.

It is common for more than one CA to exist. A quick examination of the properties of your web browser will reveal a large number of certificate authorities which your browser assumes you "trust" to perform the function of a CA. As there is more than one CA it is common for one CA to sign a digital certificate containing the public key of another CA, and vice versa, a process which is known as cross-certification.

Cross-certification is needed if more than one CA exists, since a user may not have a trusted copy of the CA's public key needed to verify another user's digital certificate. This is solved by cross-certificates, i.e. one CA's public key is signed by another CA. The user first verifies the appropriate cross-certificate, and then verifies the user certificate itself.

With many CAs one can get quite long certificate chains, as Figure 18.1 illustrates. Suppose Bob trusts the Root CA's public key and he obtains Alice's public key which is signed by the private key of CA3. He then obtains CA3's public key, either along with Alice's digital certificate or by some other means. CA3's public key comes in a certificate which is signed by the private key of CA1. Bob then needs to obtain the public key of CA1, which will be contained in a certificate signed by the Root CA. Hence, by verifying all the signatures he ends up trusting Alice's public key.

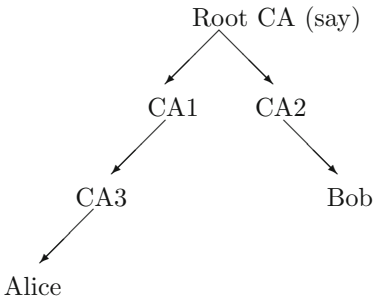


FIGURE 18.1. Example certification hierarchy

Often the function of a CA is split into two parts. One part deals with verifying the user's identity and one part actually signs the public keys. The signing is performed by the CA, whilst the identity of the user is parcelled out to a registration authority, or RA. This can be a good practice, with the CA implemented in a more secure environment to protect the long-term private key.

The main problem with a CA system arises when a user's public key is compromised or becomes untrusted for some reason. For example

- A third party has gained knowledge of the private key,
- An employee leaves the company.

As the public key is no longer to be trusted all the associated digital certificates are now invalid and need to be revoked. But these certificates can be distributed over a large number of users, each one of which needs to be told to no longer trust this certificate. The CA must somehow inform all users that the certificate(s) containing this public key is/are no longer valid, in a process called certificate revocation.

One way to accomplish this is via a Certificate Revocation List, or CRL, which is a signed statement by the CA containing the serial numbers of all certificates which have been revoked by that CA and whose validity period has not expired. One clearly need not include in this the serial numbers of certificates which have passed their expiry date. Users must then ensure they have the latest CRL. This can be achieved by issuing CRLs at regular intervals even if the list has not changed. Such a system can work well in a corporate environment when overnight background jobs are often used to make sure each desktop computer in the company is up to date with the latest software etc. For other situations it is hard to see how the CRLs can be distributed, especially if there are a large number of CAs trusted by each user.

The whole system of CAs and certificates is often called the Public Key Infrastructure, or PKI. This essentially allows a distribution of trust; the need to trust the authenticity of each individual public key in your possession is replaced by the need to trust a body, the CA, to do its job correctly.

18.2.1. Implicit Certificates: One issue with digital certificates is that they can be rather large. Each certificate needs to at least contain both the public key of the user and the signature of the certificate authority on that key. This can lead to quite large certificate sizes, as the following table demonstrates:

	RSA	DSA	EC-DSA
User's key	2024	2048	256
CA sig	2024	512	512

This assumes that for RSA keys one uses a 2048-bit modulus, for DSA one uses a 2048-bit prime p and a 256-bit prime q and for EC-DSA one uses a 256-bit curve. Hence, for example, if the CA is using 2048-bit RSA and they are signing the public key of a user using 2048-bit DSA then the total certificate size must be at least 4096 bits.

Implicit certificates enable these sizes to be reduced somewhat. An implicit certificate looks like $X|Y$ where

- X is the data being bound to the public key,
- Y is the implicit certificate on X.

From Y we need to be able to recover the public key being bound to X and implicit assurance that the certificate was issued by the CA. In the system we describe below, based on a DSA or EC-DSA, the size of Y will be 2048 or 256 bits respectively. Hence, the size of the certificate is reduced to the size of the public key being certified.

System Set-up: The CA chooses a public group G of known order n and an element $P \in G$. The CA then chooses a long-term private key c and computes the public key $Q \leftarrow P^c$. This public key should be known to all users.

Certificate Request: Suppose Alice wishes to request a certificate and the public key associated with the information ID, which could be her name. Alice computes an ephemeral secret key t and an ephemeral public key $R \leftarrow P^t$. Alice sends R and ID to the CA.

Processing of the Request: The CA checks that he wants to link ID with Alice. The CA picks another random number k and computes

$$g \leftarrow P^k R = P^k P^t = P^{k+t}.$$

Then the CA computes $s \leftarrow cH(ID\|g) + k \pmod{n}$. Then the CA sends back to Alice the pair (g, s). The implicit certificate is the pair (ID, g). We now have to convince you that, not only can Alice recover a valid public/private key pair, but also any other user can recover Alice's public key from this implicit certificate.

Alice's Key Discovery: Alice knows the following information: $t, s, R = P^t$. From this she can recover her private key via $a \leftarrow t + s \pmod{n}$. Note that Alice's private key is known only to Alice and not to the CA. In addition Alice has contributed some randomness t to her private key, as has the CA who contributed k. Her public key is then $P^a = P^{t+s} = P^t P^s = R \cdot P^s$.

User's Key Discovery: Since s and R are public, a user, say Bob, can recover Alice's public key from the above message flows via $R \cdot P^s$. But this says nothing about the linkage between the CA, Alice's public key and the ID information. Instead Bob recovers the public key from the implicit certificate (ID, g) and the CA's public key Q via the equation $P^a = Q^{H(ID\|g)}g$.

As soon as Bob sees Alice's key used in action, say he verifies a signature purported to have been made by Alice, he knows implicitly that it must have been issued by the CA, since otherwise Alice's signature would not verify correctly.

There are a number of problems with the above system which mean that implicit certificates are not used much in real life. For example:

(1) What do you do if the CA's key is compromised? Usually you pick a new CA key and re-certify the user's keys. But you cannot do this since the user's public key is chosen interactively during the certification process.
(2) Implicit certificates require the CA and users to work at the same security level. This is not considered good practice, as usually one expects the CA to work at a higher security level (say 4096-bit DSA) than the user (say 2048-bit DSA).

However for devices with restricted bandwidth implicit certificates can offer a suitable alternative where traditional certificates are not viable.

18.3. Fresh Ephemeral Symmetric Keys from Static Symmetric Keys

Recall that if we have n users each pair of whom wishes to communicate securely with each other then we would require

$$\frac{n \cdot (n-1)}{2}$$

separate static symmetric key pairs. As remarked earlier this leads to huge key management problems and issues related to the distribution of the keys. We have already mentioned that it is better to use session keys and few long-term keys, but we have not explained how one deploys the session keys.

To solve this problem a number of protocols which make use of symmetric key cryptography to distribute secret session keys have been developed, some of which we shall describe in this section. Later we shall look at public key techniques for this problem, which are often more elegant. We first need to set up some notation to describe the protocols. Firstly we set up the names of the parties and quantities involved.

- **Parties/Principals:** A, B, S.
  Assume the two parties who wish to agree a secret are A and B, for Alice and Bob. We assume that they will use a trusted third party, or TTP, which we shall denote by S.
- **Shared Secret Keys:** K_{ab}, K_{bs}, K_{as}.
  K_{ab} will denote a secret key known only to A and B.
- **Nonces:** N_a, N_b.
  Just as in Chapter 13 nonces are numbers used only once; they do not need to be random, just unique. The quantity N_a will denote a nonce originally produced by the principal A.

Note that other notations for nonces are possible and we will introduce them as the need arises.

- **Timestamps:** T_a, T_b, T_s.
  The quantity T_a is a timestamp produced by A. When timestamps are used we assume that the parties try to keep their clocks in synchronization using some other protocol.

The statement

$$A \longrightarrow B : M, A, B, \{N_a, M, A, B\}_{K_{as}}$$

means A sends to B the message to the right of the colon. The message consists of

- A nonce M,
- A the name of party A,
- B the name of party B,
- A message $\{N_a, M, A, B\}$ encrypted under the key K_{as} which A shares with S. Hence, the recipient B is unable to read the encrypted part of this message.

18.3.1. Wide-Mouth Frog Protocol: Our first protocol is the Wide-Mouth Frog protocol, which is a simple protocol invented by Burrows. The protocol transfers a key K_{ab} from A to B via S; it uses only two messages but has a number of drawbacks. In particular it requires the use of synchronized clocks, which can cause a problem in implementations. In addition the protocol assumes that A chooses the session key K_{ab} and then transports this key to user B. This implies that user A is trusted by user B to be competent in making and keeping keys secret. This is a very strong assumption and the main reason that this protocol is not used much in real life. However, it is very simple and gives a good example of how to analyse a protocol formally, which we shall come to later in this chapter. The protocol proceeds in the following steps, as illustrated in Figure 18.2,

$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$

On obtaining the first message the trusted third party S decrypts the last part of the message and checks that the timestamp is recent. This decrypted message tells S it should forward the key to the party called B. If the timestamp is verified to be recent, S encrypts the key along with its timestamp and passes this encryption on to B. On obtaining this message B decrypts the message received and checks the time stamp is recent, then he can recover both the key K_{ab} and the name A of the person who wants to send data to him using this key. The checks on the timestamps mean the session key should be recent, in that it left user A a short time ago. However, user A could have generated this key years ago and stored it on her hard disk, in which time Eve broke in and took a copy of this key.

We already said that this protocol requires that all parties need to keep synchronized clocks. However, this is not such a big problem since S checks or generates all the timestamps used in the protocol. Hence, each party only needs to record the difference between its clock and the clock owned by S. Clocks are then updated if a clock drift occurs which causes the protocol to fail. This protocol is really too simple; much of the simplicity comes by assuming synchronized clocks and by assuming party A can be trusted with creating session keys.

18.3.2. Needham–Schroeder Protocol: We shall now look at more complicated protocols, starting with one of the most famous, namely the Needham–Schroeder protocol. This protocol was developed in 1978, and is one of most highly studied protocols ever; its fame is due to the fact that even a simple protocol can hide security flaws for a long time. We shall build the protocol up slowly, starting with a simple, obviously insecure, protocol and then addressing each issue we find until we reach the final protocol. Our goal is to come up with a protocol which does not require synchronized clocks.

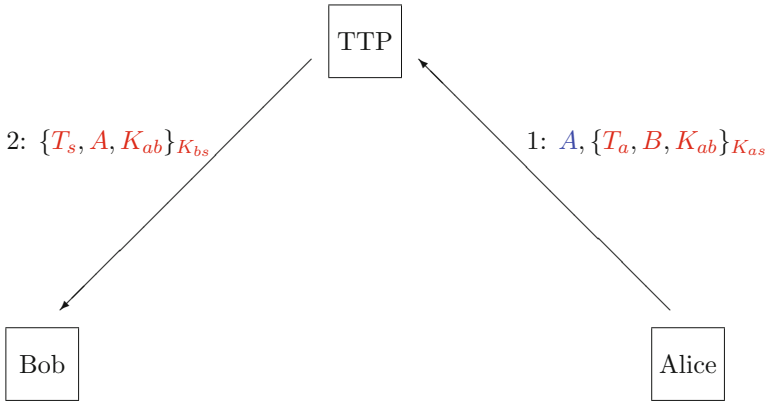


FIGURE 18.2. Wide-Mouth Frog protocol

In all of our analysis we assume that the attacker has complete control of the network; she can delay, replay and delete messages. She can even masquarade as legitimate entities, until those entities prove who they are via cryptographic means. This model of the network and attacker is called the Dolev–Yao model.

Our first simple protocol is given by the following message flows, and is illustrated in Figure 18.3,

$$A \longrightarrow S : A, B,$$
$$S \longrightarrow A : K_{ab},$$
$$A \longrightarrow B : K_{ab}, A.$$

In this protocol the trusted server S generates the key, after it is told by Alice to generate a key

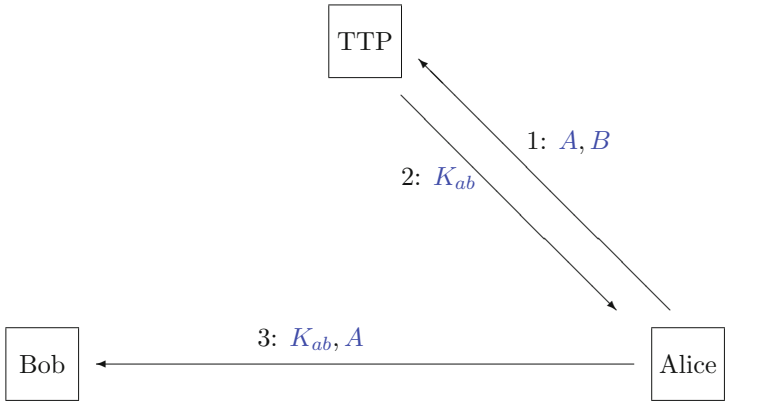


FIGURE 18.3. Protocol Version 1

for use between Alice and Bob. At the end of the protocol Bob is told by Alice that the key K_{ab} is one he should use for communication with Alice. However, it is immediately obvious that this protocol is not secure, since any eavesdropper can learn the secret key K_{ab}, since it is transmitted unencrypted.

So our first modification is to create a protocol which enables the key to remain secret, however we need to do this utilizing only the long-term static secret keys K_{as} and K_{bs}. With this modification our protocol is formed of the following message flows, and is illustrated in Figure 18.4,

$$\begin{aligned} A &\longrightarrow S : A, B, \\ S &\longrightarrow A : \{K_{ab}\}_{K_{bs}}, \{K_{ab}\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}\}_{K_{bs}}, A. \end{aligned}$$

This is slightly better, but now we notice that the attacker can take the last message from Alice

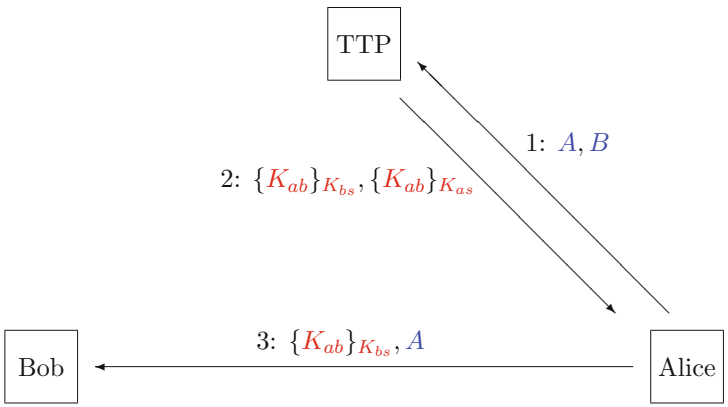


FIGURE 18.4. Protocol Version 2

to Bob, and replace it with $\{K_{ab}\}_{K_{bs}}, D$. This means that Bob will think that the key K_{ab} is for communicating with Diana and not Alice. Thus we do not have the property that Bob knows to whom he is sending information. Imagine that Bob is in love with Diana and so encrypts a message under K_{ab} saying "I love you". This message can only be decrypted by Alice, so the adversary now redirects the ciphertext to Alice. Alice decrypts the message and apparently finds out that Bob is in love with her. We easily see that this could cause problems for both Alice, Bob and Diana.

A more fundamental problem with our second protocol attempt is that there is a man-in-the-middle attack. In the following message flows, the attacker Eve (E), masquerades as both the TTP and Bob to Alice, and so is able to learn the key that Alice thinks she is using to communicate solely with Bob.

$$\begin{aligned} A &\longrightarrow E : A, B, \\ E &\longrightarrow S : A, E, \\ S &\longrightarrow E : \{K_{ae}\}_{K_{es}}, \{K_{ae}\}_{K_{as}}, \\ E &\longrightarrow A : \{K_{ae}\}_{K_{es}}, \{K_{ae}\}_{K_{as}}, \\ A &\longrightarrow E : \{K_{ae}\}_{K_{es}}, A. \end{aligned}$$

To get around these two problems we get the TTP S to encrypt the identity components, thus preventing both of the above problems. Thus our third attempt at a protocol is given by Figure 18.5 and the following message flows:

$$\begin{aligned} A &\longrightarrow S : A, B, \\ S &\longrightarrow A : \{K_{ab}, A\}_{K_{bs}}, \{K_{ab}, B\}_{K_{as}}, \\ A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}}. \end{aligned}$$

This protocol however suffers from a replay attack. Assume the attacker can determine the key

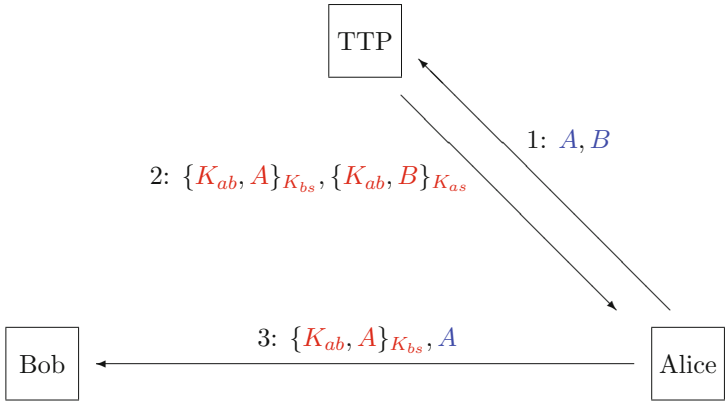


FIGURE 18.5. Protocol Version 3

K_{ab} used in an old run of the protocol; this might be because users and systems are often not as careful with respect to keys meant for short-term use (after all they *are* for short-term use!). She can then masquerade as the TTP and deliver the same responses to a new run of a protocol as to the old run, and she can do this without needing to know either K_{as} or K_{bs}.

The basic problem is that Alice does not know whether the key she receives is fresh. In the Wide-Mouth Frog protocol we guaranteed freshness by the use of synchronized clocks; we are trying to avoid them in the design of this protocol. We avoid the need to use synchronized clocks by instead utilizing nonces. This leads us to our fourth protocol attempt, given by Figure 18.6 and the following message flows:

$$
\begin{aligned}
A &\longrightarrow S : A, B, N_a, \\
S &\longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\
A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}},
\end{aligned}
$$

By Alice checking the nonce N_a received in the response from S is the same as that sent in the initial message, Alice knows that the key K_{ab} will be fresh; assuming that S is following the protocol. In this fourth variant Bob knows that Alice once was alive and sent the same ephemeral symmetric key K_{ab} as he just received. But this could have been a long while in the past. Similarly, Alice knows that only Bob could have access to K_{ab}, but she does not know whether he does or not. Thus our last modification is to add a key confirmation step, which assures both Alice and Bob that they are both alive and are able to use the key. This is achieved by adding an encrypted nonce sent by Bob, which Alice decrypts, modifies and sends back to Bob encrypted. With this change we get the Needham–Schroeder protocol, see Figure 18.7, and the message flows:

$$
\begin{aligned}
A &\longrightarrow S : A, B, N_a, \\
S &\longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}, \\
A &\longrightarrow B : \{K_{ab}, A\}_{K_{bs}}, \\
B &\longrightarrow A : \{N_b\}_{K_{ab}}, \\
A &\longrightarrow B : \{N_b - 1\}_{K_{ab}}.
\end{aligned}
$$

We now look again at each message in detail, and summarize what it achieves:

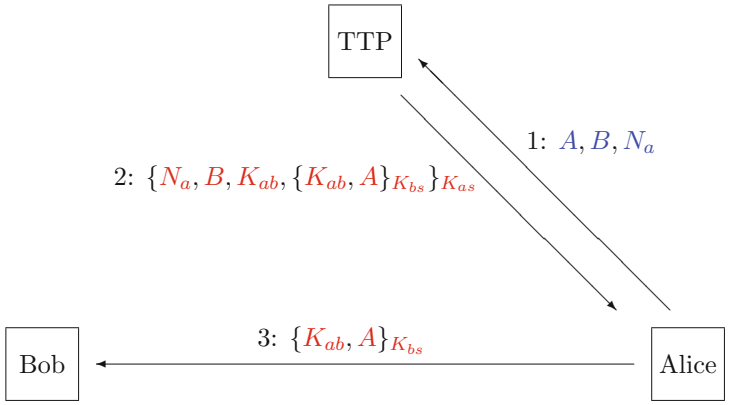


FIGURE 18.6. Protocol Version 4

- The first message tells S that Alice wants a key to communicate with Bob.
- In the second message S generates the session key K_{ab} and sends it back to Alice. The nonce N_a is included so that Alice knows this was sent after her request of the first message. The session key is also encrypted under the key K_{bs} for sending to Bob.
- The third message conveys the session key to Bob.
- Bob needs to check that the third message was not a replay. So he needs to know whether Alice is still alive; hence, in the fourth message he encrypts a nonce back to Alice.
- In the final message, to prove to Bob that she is still alive, Alice encrypts a simple function of Bob's nonce back to Bob.

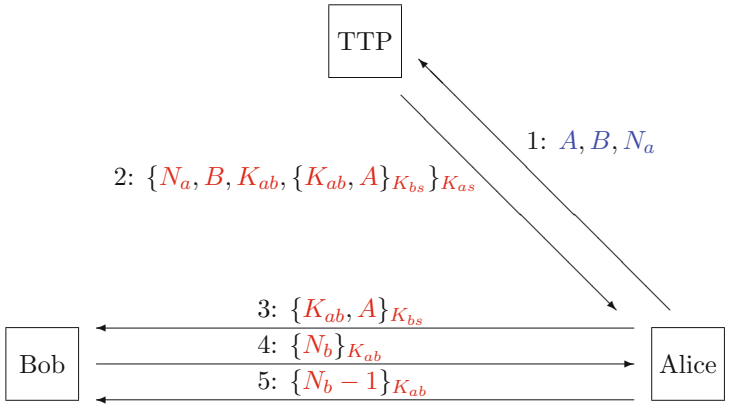


FIGURE 18.7. Needham–Schroeder protocol

The main problem with the Needham–Schroeder protocol is that Bob does not know that the key he shares with Alice is fresh, a fact which was not spotted until some time after the original protocol was published. An adversary who finds an old session transcript can, after finding the old session key by some other means, use the old session transcript in the last three messages involving Bob. Hence, the adversary can get Bob to agree to a key with the adversary, which Bob thinks he is sharing with Alice.

Note that Alice and Bob have their secret session key generated by the TTP and so neither party needs to trust the other to produce "good" keys. They of course trust the TTP to generate good keys since the TTP is an authority trusted by everyone. In some applications this last assumption is not valid and more involved algorithms, or public key algorithms, are required.

18.3.3. Kerberos: We end this section by looking at Kerberos. Kerberos is an authentication system based on symmetric encryption, with keys shared with an authentication server; it is based on ideas underlying the Needham–Schroeder protocol. Kerberos was developed at MIT around 1987 as part of Project Athena. A modified version of this original version of Kerberos is now used in many versions of the Windows operating system, and in many other systems.

The network is assumed to consist of clients and a server, where the clients may be users, programs or services. Kerberos keeps a central database of clients including a secret key for each client, hence Kerberos requires a key space of size $O(n)$ if we have n clients. Kerberos is used to provide authentication of one entity to another and to issue session keys to these entities.

In addition Kerberos can run a ticket-granting system to enable access control to services and resources. This division mirrors what happens in real companies. For example, in a company the personnel department administers who you are, whilst the computer department administers what resources you can use. This division is also echoed in Kerberos with an authentication server and a ticket generation server TGS. The TGS grants tickets to enable users to access resources, such as files, printers, etc.

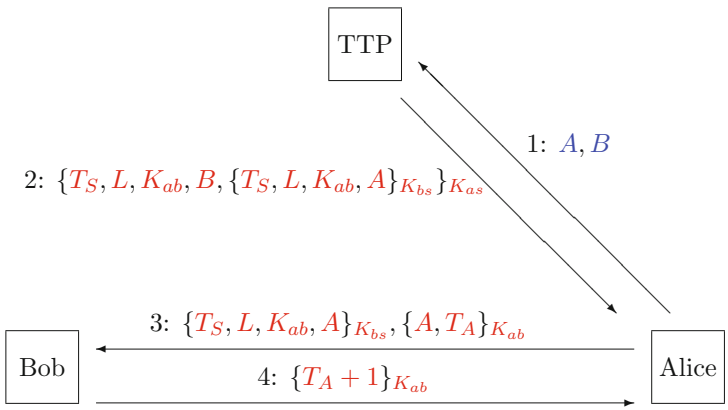


FIGURE 18.8. Kerberos

Suppose A wishes to access a resource B. First A logs in to the authentication server using a password. The user A is given a ticket from this server encrypted under her password. This ticket contains a session key K_{as}. She now uses K_{as} to obtain a ticket from the TGS S to access the resource B. The output of the TGS is a key K_{ab} , a timestamp T_S and a lifetime L. The output of the TGS is used to authenticate A in subsequent traffic with B. The flows look something like those given in Figure 18.8,

$$
\begin{aligned}
&A \longrightarrow S : A, B,\\
&S \longrightarrow A : \{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}},\\
&A \longrightarrow B : \{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}},\\
&B \longrightarrow A : \{T_A + 1\}_{K_{ab}}.
\end{aligned}
$$

Again we describe what each message flow is trying to achieve:

- The first message is A telling S that she wants to access B.
- If S allows this access then a ticket $\{T_S, L, K_{ab}, A\}$ is created. This is encrypted under K_{bs} and sent to A for forwarding to B. The user A also gets a copy of the key in a form readable by her.
- The user A wants to verify that the ticket is valid and that the resource B is alive. Hence, she sends an encrypted nonce/timestamp T_A to B.
- The resource B sends back the encryption of $T_A + 1$, after checking that the timestamp T_A is recent, thus proving that he knows the key and is alive.

Thus we have removed the problems associated with the Needham–Schroeder protocol by using timestamps, but this has created a requirement for synchronized clocks.

18.4. Fresh Ephemeral Symmetric Keys from Static Public Keys

Recall that the main drawback with the use of fast bulk encryption based on block or stream ciphers was the problem of key distribution. We have already seen a number of techniques to solve this problem, using protocols which are themselves based on symmetric key techniques. These, however, also have problems associated with them. For example, the symmetric key protocols required the use of already deployed long-term keys between each user and a trusted central authority. In this section we look at two public-key-based techniques. The first, called *key transport*, uses public key encryption to transmit a symmetric key from one user to the other; the second, called *key agreement*, is a protocol which as output produces a symmetric key, and which uses public key signatures to authenticate the parties.

18.4.1. Key Transport: Let $(e_{\mathfrak{pk}}, d_{\mathfrak{sk}})$ be a public key encryption scheme with public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$ associated with a user Bob, via a certificate. Suppose Alice wants to send a symmetric key over to Bob, she first looks up Bob's public key in a directory, she generates the required symmetric key K_{ab} from the required space and then encrypts this and sends it to Bob. This is a very simple scheme whose flow is given pictorially in Figure 18.9 and in symbols by

$$A \longrightarrow B : e_{\mathfrak{pk}}(K_{ab}).$$

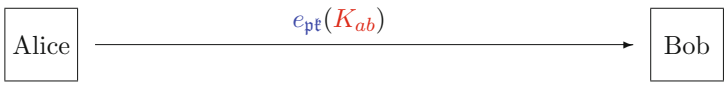


FIGURE 18.9. Public-key-based key transport: Version 1

This protocol is very much like the first part of a hybrid encryption scheme. However, it does not achieve all that we require. Firstly, whilst Alice knows that only Bob can decrypt the ciphertext to obtain the new key K_{ab}, Bob does not know that the ciphertext came from Alice. So Bob does not know to whom the key K_{ab} should be associated. One way around this is for Alice to also append a digital signature to the ciphertext. So now we have two public/private key pairs: a signature pair for Alice $(\mathfrak{pk}_A, \mathfrak{sk}_A)$ for some public key signature algorithm Sig, and an encryption pair for Bob $(\mathfrak{pk}_B, \mathfrak{sk}_B)$ for some encryption algorithm $e_{\mathfrak{pk}_B}$. The resulting protocol flow is given in Figure 18.10. However, this protocol suffers from another weakness; we can get Bob to think he is sharing a key with Eve, when he is actually sharing a key with Alice. The attack goes as follows:

$$A \longrightarrow E : c := e_{\mathfrak{pk}_B}(K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_A}(c)$$
$$E \longrightarrow B : c := e_{\mathfrak{pk}_B}(K_{ab}), \mathsf{Sig}_{\mathfrak{sk}_E}(c).$$

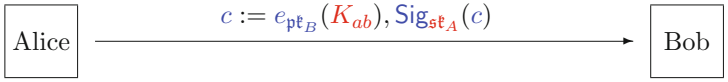


FIGURE 18.10. Public-key based key transport: Version 2

This works because the identity of the sender is not bound to the ciphertext. Following the lead from our examples in Section 18.3, the simple solution to this problem is to encrypt the sender's identity along with the transmitted key, as in Figure 18.11. But even this cannot be considered secure due to a replay attack which we will outline later.

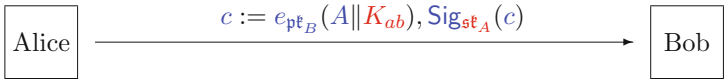


FIGURE 18.11. Public-key based key transport: Version 3

Forward Secrecy: All of the systems based on symmetric key encryption given in Section 18.3, and the previous method of key transport using public key-encryption-based key transport are not *forward secure*. A system is said to have forward secrecy if compromising of a long-term private key, i.e. $\mathfrak{sk}_B$ in the above protocol, at some point, in the future does not compromise the security of communications made using that key in the past. Notice in Figure 18.11 that if the recipient's private key $\mathfrak{sk}_B$ is compromised in the future, then all communications in the past are also compromised.

In addition using key transport implies that the recipient trusts the sender to be able to generate, in a sensible way, the session key. Sometimes the recipient may wish to contribute some randomness of their own to the session key. However, this can only be done if both parties are online at the same moment in time. Key transport is thus more suited to the case where only the sender is online, as in applications like email, for example.

The idea of both parties contributing to the entropy of the session key not only aids in creating perfectly secure schemes, it also avoids the replay attack we have on our key transport protocol. The next set of protocols, based on Diffie–Hellman key exchange, does indeed contribute entropy from both sides.

18.4.2. Diffie–Hellman Key Exchange: To avoid the fact that key transport based on public key encryption is not forward secure, and the problem of one party generating the key, the modern way of using public key techniques to create symmetric keys between two parties is based on a process called *key exchange*.

Key exchange was introduced in the same seminal paper by Diffie and Hellman in which they introduced public key cryptography. Their protocol for key distribution, called *Diffie–Hellman key exchange*, allows two parties to agree a secret key over an insecure channel without having met before. Its security is based on the discrete logarithm problem in a finite abelian group G of prime order q. In the original paper the group is taken to be a subgroup G of $\mathbb{F}_p^*$, but now more efficient versions can be produced by taking G to be a subgroup of an elliptic curve, in which case the protocol is called EC-DH.

In Diffie–Hellman the two parties each have their own ephemeral secrets a and b, which are elements in the group $\mathbb{Z}/q\mathbb{Z}$. The basic message flows for the Diffie–Hellman protocol are given by

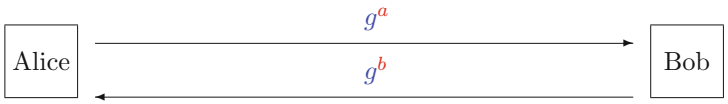


FIGURE 18.12. Diffie–Hellman key exchange

Figure 18.12 and the following notational representation:

$$A \longrightarrow B : \mathfrak{ek}_A = g^a,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^b.$$

From these exchanged ephemeral key values, $\mathfrak{ek}_A$ and $\mathfrak{ek}_B$, and their respective ephemeral secret keys, a and b, both parties can agree on the same secret session key:

- Alice can compute $K \leftarrow {\mathfrak{ek}_B}^a = (g^b)^a$, since she knows a and was sent $\mathfrak{ek}_B = g^b$ by Bob,
- Bob can also compute $K \leftarrow {\mathfrak{ek}_A}^b = (g^a)^b$, since he knows b and was sent $\mathfrak{ek}_A = g^a$ by Alice.

Eve, the attacker, can see the messages g^a and g^b and then needs to recover the secret key $K = g^{a \cdot b}$ which is exactly the Diffie–Hellman problem considered in Chapter 3. Hence, the security of the above protocol rests not on the difficulty of solving the discrete logarithm problem, DLP, but on the difficulty of solving the Diffie–Hellman problem, DHP. Recall that it may be the case that it is easier to solve the DHP than the DLP, although no one believes this to be true for the groups that are currently used in real-life protocols.

In practice one does not want the agreed key to be an element of a group, one requires it to be a bit string of a given length. Hence, in real systems one applies a key derivation function to the agreed secret group element, to obtain the required key for future use. It turns out that not only is this operation important for functional reasons, but in addition when we model the KDF as a random oracle one can prove the security of Diffie–Hellman-related protocols in the random oracle model. Thus we almost always use the derived key $k = H(K)$, for some random oracle H.

Notice that the Diffie–Hellman protocol can be performed both online (in which case both parties contribute to the randomness in the shared session key) or offline, where one of the parties uses a long-term key of the form g^a instead of an ephemeral key. Hence, the Diffie–Hellman protocol can be used as a key exchange or as a key transport protocol. We shall focus however on its use as a key exchange protocol.

Finite Field Example: The following is a very small example; we let the domain parameters be given by

$$p = 2\,147\,483\,659, \quad q = 2\,402\,107, \quad \text{and} \quad g = 509\,190\,093,$$

but in real life one would take $p \approx 2^{2048}$. Note that g has prime order q in the field $\mathbb{F}_p$. The following diagram indicates a possible message flow for the Diffie–Hellman protocol:

Alice		Bob
$a = 12\,345$		$b = 654\,323$,
$\mathfrak{ek}_A = g^a = 382\,909\,757$	$\longrightarrow$	$\mathfrak{ek}_A = 382\,909\,757$,
$\mathfrak{ek}_B = 1\,190\,416\,419$	$\longleftarrow$	$\mathfrak{ek}_B = g^b = 1\,190\,416\,419$.

The shared secret group element is then computed via

$${\mathfrak{ek}_A}^b = 382\,909\,757^{654\,323} \pmod p = 881\,311\,606,$$
$${\mathfrak{ek}_B}^a = 1\,190\,416\,419^{12\,345} \pmod p = 881\,311\,606,$$

with the actual secret key being given by $k = H(881\,311\,606)$ for some KDF H, which we model as a random oracle.

Notice that group elements are transmitted in the protocol, hence when using a finite field such as $\mathbb{F}_p^*$ for the Diffie–Hellman protocol the communication costs are around 2048 bits in each direction, since it is prudent to choose $p \approx 2^{2048}$. However, when one uses an elliptic curve group $E(\mathbb{F}_q)$ one can choose $q \approx 2^{256}$, and so the communication costs are much less, namely around 256 bits in each direction. In addition the group exponentiation step for elliptic curves can be done more efficiently than that for finite prime fields.

Elliptic Curve Example: As a baby example of EC-DH consider the elliptic curve

$$E : Y^2 = X^3 + X - 3$$

over the field $\mathbb{F}_{199}$. Let the base point be given by $G = (1, 76)$, which has prime order $q = \#E(\mathbb{F}_{199}) = 197$. Then a possible EC-DH message flow is given by

Alice		Bob
$a = 23$		$b = 86$,
$\mathfrak{ek}_A = [a]G = (2, 150)$	$\longrightarrow$	$\mathfrak{ek}_A = (2, 150)$,
$\mathfrak{ek}_B = (123, 187)$	$\longleftarrow$	$\mathfrak{ek}_B = [b]G = (123, 187)$.

The shared secret key is then computed via

$$[b]\mathfrak{ek}_A = [86](2, 150) = (156, 75),$$
$$[a]\mathfrak{ek}_B = [23](123, 187) = (156, 75).$$

The shared key is then usually taken to be the x-coordinate 156 of the computed point. In addition, instead of transmitting the points, we transmit the compression of the point, which results in a significant saving in bandwidth.

18.4.3. Signed Diffie–Hellman: So we seem to have solved the key distribution problem. But there is an important problem: you need to be careful *who* you are agreeing a key with. Alice has no assurance that she is agreeing a key with Bob, which can lead to the following man-in-the-middle attack:

Alice		Eve		Bob
a	$\longrightarrow$	g^a,		
g^m	$\longleftarrow$	m,		
g^{am}		g^{am},		
		n	$\longrightarrow$	g^n,
		g^b	$\longleftarrow$	b,
		g^{bn}		g^{bn}.

In the man-in-the-middle attack

- Alice agrees a key with Eve, thinking it is Bob with whom she is agreeing a key with,
- Bob agrees a key with Eve, thinking it is Alice,
- Eve can now examine communications as they pass through her i.e. she acts as a router. She does not alter the plaintext, so her actions go undetected.

So we can conclude that the Diffie–Hellman protocol on its own is not enough. For example how does Alice know with whom she is agreeing a key? Is it Bob or Eve? One way around the man-in-the-middle attack on the Diffie–Hellman protocol is for Alice to sign her message to Bob and Bob to sign his message to Alice. In this way both parties know who they are talking to. This produces the protocol called *signed-Diffie–Hellman* given in Figure 18.13 and with message flows:

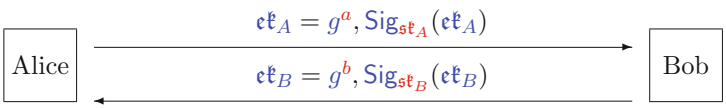


FIGURE 18.13. Signed Diffie–Hellman key exchange

$$A \longrightarrow B : \mathfrak{cert}_A = g^a, \mathsf{Sig}_{\mathfrak{st}_A}(\mathfrak{cert}_A),$$
$$B \longrightarrow A : \mathfrak{cert}_B = g^b, \mathsf{Sig}_{\mathfrak{st}_B}(\mathfrak{cert}_B).$$

The problem is that we again have the attack of stripping the signature from Alice's message, replacing it with Eve's; then Bob will think he shares a key with Eve, whereas actually he shares a key with Alice. In Figure 18.11 this was solved by encrypting the identity of the sender, so that it could not be tampered with. However, in Diffie–Hellman key exchange there is no encryption used into which we can embed the identity.

18.4.4. Station-to-Station Protocol: To get around this latter problem the following protocol was invented, called the *station-to-station* (STS) protocol. The original presentation dates back to 1987. The basic idea is that the two parties encrypt their signatures using some symmetric key encryption algorithm (e_k, d_k), and the key, $k \leftarrow H(g^{a \cdot b})$, derived from the key exchange protocol. In particular this means that the initiator's, in our case Alice's, signature needs to be sent in a third message flow. Thus we have the flows:

$$\begin{aligned} &A \longrightarrow B : \mathfrak{cert}_A = g^a, \\ &B \longrightarrow A : \mathfrak{cert}_B = g^b, e_k\left(\mathsf{Sig}_{\mathfrak{st}_B}(\mathfrak{cert}_B, \mathfrak{cert}_A)\right) && \text{where } k \leftarrow H({\mathfrak{cert}_A}^b), \\ &A \longrightarrow B : e_k\left(\mathsf{Sig}_{\mathfrak{st}_A}(\mathfrak{cert}_A, \mathfrak{cert}_B)\right) && \text{where } k \leftarrow H({\mathfrak{cert}_B}^a). \end{aligned}$$

Notice that the messages signed by each party have the group elements in different orders. Another variant on the STS protocol is for the parties to authenticate their signatures using a MAC function $(\mathsf{Mac}_{k'}, \mathsf{Verify}_{k'})$. In this variant the key derivation function is used to derive a key k to use in the following protocol (for which we require key agreement scheme), and a separate key k' to perform the authentication of the signature values. The reason for this is to provide a clear separation between k and the protocol used to derive it. So in this variant of the STS protocol the message flows become:

$$\begin{aligned} &A \longrightarrow B : \mathfrak{cert}_A = g^a, \\ &B \longrightarrow A : \mathfrak{cert}_B = g^b, S_B := \mathsf{Sig}_{\mathfrak{st}_B}(\mathfrak{cert}_B, \mathfrak{cert}_A), \mathsf{Mac}_{k'}(S_B) && \text{where } k\|k' \leftarrow H({\mathfrak{cert}_A}^b) \\ &A \longrightarrow B : S_A := \mathsf{Sig}_{\mathfrak{st}_A}(\mathfrak{cert}_A, \mathfrak{cert}_B), \mathsf{Mac}_{k'}(S_A) && \text{where } k\|k' \leftarrow H({\mathfrak{cert}_B}^a) \end{aligned}$$

18.4.5. Blake-Wilson–Menezes Protocol: One can ask whether one can obtain authentication without the need for signatures and/or MACs as in the station-to-station protocol, and without the need for additional data to be sent, or the additional third message flow. The answer to all of these questions is *yes*, as the following protocol, due to Blake-Wilson and Menezes, and the MQV protocol of the next section will show.

Originally the Diffie–Hellman protocol was presented as a way to provide authentic keys given shared static public keys $\mathfrak{pk}_A = g^a$ and $\mathfrak{pk}_B = g^b$. In other words these values were not exchanged,

but used to produce a static authenticated shared symmetric key. Then people realized by exchanging ephemeral versions of such keys one could obtain new symmetric keys for each iteration. It turns out that we can combine both the static and the public variants of Diffie–Hellman key exchange to obtain an authenticated key agreement protocol, without the need for digital signatures.

To do this we assume that Alice has a long-term *static* public/private key pair given by $(\mathfrak{pk}_A = g^a, a)$, and Bob has a similar long-term public/private key pair given by $(\mathfrak{pk}_B = g^b, b)$. We first assume that Alice has obtained an authentic version of Bob's public key $\mathfrak{pk}_B$, say via a digital certificate, and vice versa. We can then obtain an authenticated key agreement protocol using only two message flows, as follows:

$$A \longrightarrow B : \mathfrak{ek}_A = g^x,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^y.$$

Notice that the message flows are *identical* to the original Diffie–Hellman protocol. The key difference is in how the shared secret key k is derived. Alice derives it via the equation

$$k \leftarrow H({\mathfrak{pk}_B}^x, {\mathfrak{ek}_B}^a) = H(g^{b\cdot x}, g^{y\cdot a}).$$

The same key is derived by Bob using the equation

$$k \leftarrow H({\mathfrak{ek}_A}^b, {\mathfrak{pk}_A}^y) = H(g^{x\cdot b}, g^{a\cdot y}).$$

18.4.6. MQV Protocol: The only problem with the previous protocol is that one must perform three exponentiations per key agreement per party. Alice needs to compute $\mathfrak{ek}_A = g^x$, ${\mathfrak{ek}_B}^x$ and ${\mathfrak{pk}_B}^a$. This led Menezes, Qu and Vanstone to invent the following protocol, called the MQV protocol. Once again, being based on the Diffie–Hellman protocol, security is based on the discrete logarithm problem in a group G generated by g. Like the Blake-Wilson–Menezes protocol, MQV works by assuming that both parties, Alice and Bob, first generate long-term public/private key pairs which we shall denote by $(\mathfrak{pk}_A = g^a, a)$ and $(\mathfrak{pk}_B = g^b, b)$. Again, we shall assume that Bob knows that $\mathfrak{pk}_A$ is the authentic public key belonging to Alice and that Alice knows that $\mathfrak{pk}_B$ is the authentic public key belonging to Bob.

Assume Alice and Bob now want to agree on a secret session key; they execute the same message flows as the Blake-Wilson–Menezes protocol, namely:

$$A \longrightarrow B : \mathfrak{ek}_A = g^x,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^y.$$

So this does not look much different to the standard un-authenticated Diffie–Hellman protocol or the Blake-Wilson–Menezes protocol. However, the key difference is in how the final session key is created from the relevant values.

Assume you are Alice, so you know

$$\mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B, a \text{ and } x.$$

Let l denote half the bit size of the order of the group G, for example if we are using a group with order $q \approx 2^{256}$ then we set $l = 256/2 = 128$. To determine the session key, Alice now computes

(1) Convert $\mathfrak{ek}_A$ to an integer i.
(2) $s_A \leftarrow (i \pmod{2^l}) + 2^l$.
(3) Convert $\mathfrak{ek}_B$ to an integer j.
(4) $t_A \leftarrow (j \pmod{2^l}) + 2^l$.
(5) $h_A \leftarrow x + s_A \cdot a \pmod{q}$.
(6) $K_A \leftarrow (\mathfrak{ek}_B \cdot {\mathfrak{pk}_B}^{t_A})^{h_A}$.

Notice how s_A and t_A are exactly l bits in length and, assuming the conversion of a group element to an integer results in a "random-looking" integer, these values will also behave like random l-bit integers. Thus Alice needs to compute one full exponentiation, to produce $\mathfrak{ek}_A$, and one multi-exponentiation to produce K_A.

Bob runs exactly the same calculation but with the roles of the public and private keys swapped around in the obvious manner, namely:

(1) Convert $\mathfrak{ek}_B$ to an integer i.
(2) $s_B \leftarrow (i \pmod{2^l}) + 2^l$.
(3) Convert $\mathfrak{ek}_A$ to an integer j.
(4) $t_B \leftarrow (j \pmod{2^l}) + 2^l$.
(5) $h_B \leftarrow y + s_B \cdot b \pmod{q}$.
(6) $K_B \leftarrow (\mathfrak{ek}_A \cdot {\mathfrak{pk}_A}^{t_B})^{h_B}$.

Then $K_A = K_B$ is the shared secret. To see why the K_A computed by Alice and the K_B computed by Bob are the same we notice that the s_A and t_A seen by Alice, are swapped when seen by Bob, i.e. $s_A = t_B$ and $s_B = t_A$. We see that

$$\begin{aligned}
\mathrm{dlog}_g(K_A) &= \mathrm{dlog}_g\left((\mathfrak{ek}_B \cdot {\mathfrak{pk}_B}^{t_A})^{h_A}\right) = (y + b \cdot t_A) \cdot h_A \\
&= y \cdot (x + s_A \cdot a) + b \cdot t_A \cdot (x + s_A \cdot a) = y \cdot (x + t_B \cdot a) + b \cdot s_B \cdot (c + t_B \cdot a) \\
&= x \cdot (y + s_B \cdot b) + a \cdot t_B \cdot (d + s_B \cdot b) = (x + a \cdot t_B) \cdot h_B \\
&= \mathrm{dlog}_g\left((\mathfrak{ek}_A \cdot \mathfrak{pk}_A^{t_B})^{h_B}\right) = \mathrm{dlog}_g(K_B).
\end{aligned}$$

18.5. The Symbolic Method of Protocol Analysis

One can see that the above protocols are very intricate; spotting flaws in them can be a very subtle business. A number of different approaches have been proposed to try and make the design of these protocols more scientific. The first school is based on so-called formal methods, and treats protocols via means of a symbolic algebra. The second school is closer to our earlier modelling of encryption and signature schemes, in that it is based on a cryptographic game between a challenger and an adversary.

The most influential of the methods in the first school is the BAN logic invented by Burrows, Abadi and Needham. The BAN logic has a large number of drawbacks compared to more modern logical analysis tools, but was very influential in the design and analysis of symmetric-key-based key agreement protocols such as Kerberos and the Needham–Schroeder protocol. It has now been supplanted by more complicated logics and formal methods, but it is of historical importance and the study of the BAN logic can still be very instructive for protocol designers. The main benefit in using symbolic methods and logics is that the analysis can usually be semi-automated via theorem provers and the like; this should be compared to the cryptographic analysis method which is still done mainly by hand.

First we really need to pause and decide what are the goals of key agreement and key transport, and what position the parties start from. In the symmetric key setting we assume all parties, A and B say, only share secret keys K_{as} and K_{bs} with the trusted third party S. In the public key setting we assume that all parties have a public/private key pair $(\mathfrak{pk}_A, \mathfrak{sk}_A)$, and that the public key $\mathfrak{pk}_A$ is bound to an entity's identity A, via some form of certificate.

In both cases parties A and B want to agree and/or transport a symmetric session key K_{ab} for use in some further protocol. This new session key should be fresh, i.e. it has not been used by any other party before and has been recently created. The freshness property will stop attacks whereby the adversary replays messages so as to use an old key again. Freshness can also be useful in deducing that the party with which you are communicating is still alive.

We also need to decide what capabilities an attacker has. As always we assume the worst possible situation in which an attacker can intercept any message flow over the network. She can then stop a message, alter it or change its destination. An attacker is also able to distribute her own messages over the network. With such a high-powered attacker it is often assumed that the attacker *is* the network.

The main idea of BAN logic is that one should concentrate on what the parties believe is happening. It does not matter what is actually happening; we need to understand exactly what each party can logically deduce, from its own view of the protocol, about what is actually happening.

In the BAN logic, complex statements are made up of some basic atomic statements which are either true or false. These atomic statements can be combined into more complex ones using conjunction, which is denoted by a comma. The basic atomic statements are given by:

- $P| \equiv X$ means P *believes* (or is entitled to believe) X.
  The principal P may act as though X is true.
- $P \lhd X$ means P *sees* X.
  Someone has sent a message to P containing X, so P can now read and repeat X.
- $P| \sim X$ means P *once said* X and P believed X when it was said.
  Note this tells us nothing about whether X was said recently or in the distant past.
- $P| \Rightarrow X$ means P has *jurisdiction* over X.
  This means P is an authority on X and should be trusted on this matter.
- $\#X$ means the formula X is *fresh*.
  This is usually used for nonces.
- $P \overset{K}{\leftrightarrow} Q$ means P and Q may use the *shared key* K to communicate.
  The key is assumed good and it will never be discovered by anyone other than P and Q, unless the protocol itself makes this happen.
- $\{X\}_K$, means as usual that X is *encrypted* under the key K.
  The encryption is assumed to be perfect in that X will remain secret unless deliberately disclosed by a party at some other point in the protocol.

We start with a set of statements which are assumed to be true at the start of the protocol. When executing the protocol we infer the truth of new statements via the BAN logic postulates, or rules of inference. The format we use to specify rules of inference is as follows:

$$\frac{A, B}{C}$$

which means that if A and B are true then we can conclude C is also true. This is a standard notation used in many areas of logic within computer science.

- **Message Meaning Rule:**

$$\frac{A| \equiv A \overset{K}{\leftrightarrow} B, A \lhd \{X\}_K}{A| \equiv B| \sim X}.$$

  In words, if both
  - A believes she shares the key K with B,
  - A sees X encrypted under the key K,

  we can deduce that A believes that B once said X. Note that this implicitly assumes that A never said X.
- **Nonce Verification Rule:**

$$\frac{A| \equiv \#X, A| \equiv B| \sim X}{A| \equiv B| \equiv X}.$$

  In words, if both
  - A believes X is fresh (i.e. recent),

– A believes B once said X,

then we can deduce that A believes that B still believes X.

- **Jurisdiction Rule:**
$$\frac{A| \equiv B| \Rightarrow X, A| \equiv B| \equiv X}{A| \equiv X}.$$
In words, if both
  – A believes B has jurisdiction over X, i.e. A trusts B on X,
  – A believes B believes X,

  then we conclude that A also believes X.
- **Other Rules:** The belief operator and conjunction can be manipulated as follows:
$$\frac{P| \equiv X, P| \equiv Y}{P| \equiv (X, Y)}, \quad \frac{P| \equiv (X, Y)}{P| \equiv X}, \quad \frac{P| \equiv Q| \equiv (X, Y)}{P| \equiv Q| \equiv X}.$$
A similar rule also applies to the "once said" operator
$$\frac{P| \equiv Q| \sim (X, Y)}{P| \equiv Q| \sim X}.$$
Note that $P| \equiv Q| \sim X$ and $P| \equiv Q| \sim Y$ does not imply $P| \equiv Q| \sim (X, Y)$, since that would imply X and Y were said at the same time. Finally, if part of a formula is fresh then so is the whole formula
$$\frac{P| \equiv \#X}{P| \equiv \#(X, Y)}.$$

We wish to analyse a key agreement protocol between A and B using the BAN logic. But what is the goal of such a protocol when expressed in this formalism? The minimum we want to achieve is

$$A| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A \overset{K}{\leftrightarrow} B,$$

i.e. both parties believe they share a secret key with each other. However, we could expect to achieve more, for example

$$A| \equiv B| \equiv A \overset{K}{\leftrightarrow} B \text{ and } B| \equiv A| \equiv A \overset{K}{\leftrightarrow} B,$$

which is called key confirmation. In words, we may want to achieve that, after the protocol has run, A is assured that B knows he is sharing a key with A, and it is the same key A believes she is sharing with B.

Before analysing a protocol using the BAN logic we convert the protocol into logical statements. This process is called *idealization*, and is the most error prone part of the procedure since it cannot be automated. We also need to specify the assumptions, or axioms, which hold at the beginning of the protocol. To see this process in "real life" we analyse the Wide-Mouth Frog protocol for key agreement using synchronized clocks.

Example: Wide-Mouth Frog Protocol: Recall the Wide-Mouth Frog protocol

$$A \longrightarrow S : A, \{T_a, B, K_{ab}\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A, K_{ab}\}_{K_{bs}}.$$

This becomes the idealized protocol

$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}},$$
$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$

One should read the idealization of the first message as telling S that

- T_a is a timestamp/nonce,
- K_{ab} is a key which is meant as a key to communicate with B.

So what assumptions exist at the start of the protocol? Clearly A, B and S share secret keys, which in BAN logic becomes

$$A| \equiv A \overset{K_{as}}{\leftrightarrow} S, \qquad S| \equiv A \overset{K_{as}}{\leftrightarrow} S,$$
$$B| \equiv B \overset{K_{bs}}{\leftrightarrow} S, \qquad S| \equiv B \overset{K_{bs}}{\leftrightarrow} S.$$

There are a couple of nonce assumptions,

$$S| \equiv \#T_a \text{ and } B| \equiv \#T_s.$$

Finally, we have the following three assumptions

- B trusts A to invent good keys,

$$B| \equiv (A| \Rightarrow A \overset{K_{ab}}{\leftrightarrow} B),$$

- B trusts S to relay the key from A,

$$B| \equiv (S| \Rightarrow A| \equiv A \overset{K_{ab}}{\leftrightarrow} B),$$

- A knows the session key in advance,

$$A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

Notice how these last three assumptions specify the problems we associated with this protocol in the earlier section. Using these assumptions we can now analyse the protocol.

- Let us see what we can deduce from the first message

$$A \longrightarrow S : \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}}.$$

  - Since S sees the message encrypted under K_{as} he can deduce that A said the message.
  - Since T_a is believed by S to be fresh he concludes the whole message is fresh.
  - Since the whole message is fresh, S concludes that A currently believes the whole of it.
  - S then concludes

$$S| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

    which is what we need to conclude so that S can send the second message of the protocol.

- We now look at what happens when we analyse the second message

$$S \longrightarrow B : \{T_s, A| \equiv A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}.$$

  - Since B sees the message encrypted under K_{bs} he can deduce that S said the message.
  - Since T_s is believed by B to be fresh he concludes the whole message is fresh.
  - Since the whole message is fresh, B concludes that S currently believes the whole of it.
  - So B believes that S believes the second part of the message.
  - But B believes S has authority on whether A knows the key and B believes A has authority to generate the key.

- From the analysis of both messages we can conclude

$$B| \equiv A \overset{K_{ab}}{\leftrightarrow} B$$

  and

$$B| \equiv A| \equiv A \overset{K_{ab}}{\leftrightarrow} B.$$

  Combining this with our axiom, $A| \equiv A \overset{K_{ab}}{\leftrightarrow} B$, we conclude that the key agreement protocol is sound. The only requirement we have not met is that

$$A| \equiv B| \equiv A \overset{K_{ab}}{\leftrightarrow} B,$$

i.e. A does not achieve confirmation that B has received the key.

Notice what the application of the BAN logic has done is to make the axioms clearer, so it is easier to compare which assumptions each protocol needs to make it work. In addition it clarifies what the result of running the protocol is from all parties' points of view. However, it does not *prove* the protocol is secure in a cryptographic sense. To do that we need a more complex method of analysis, which is less amenable to computer application.

18.6. The Game-Based Method of Protocol Analysis

To provide a more rigorous analysis of protocols (which does not assume, for example, that encryption works as a perfect black box), we now present a method of analysis which resembles the games we used to introduce encryption, signatures and MACs. Recall that there we had an adversary A who tried to achieve a certain goal, using a set of powers. The goal was presented as some winning condition, and the powers were presented as giving the adversary access to some oracles. For key agreement protocols we will adopt a winning condition which is akin to the Real-or-Random winning condition for the encryption games, i.e. an adversary should not be able to tell the difference between a real key agreed during a key agreement protocol and a key chosen completely at random.

What is more complicated is the definition of the oracles. In our previous examples the oracles were relatively simple; they either encrypted, decrypted, signed or verified some data given some hidden key. As a key agreement protocol is a *protocol*, data is passed between players; in addition there could be many keys within the protocol and so one piece of data could be sent to be processed by different entities using different keys (recall we assume the adversary has control of the network). In addition we have seen attacks on key agreement protocols which rely on messages being passed to additional entities, and not just two. Thus we need to model security where there are many participants. In addition participants may be interacting with many parties at the same time, and could be interacting with the same party many times (e.g. a client connecting to a web server in multiple sessions).

Modelling the Participants: We first set up some parties. These are going to be users $U \in \mathcal{U}$ who start our game being honest participants in the protocol. For symmetric key based protocols each entity will have a list of secret keys k_{US} of keys shared with a special trusted party S. For public key protocols each party will have a public/private key pair $(\mathfrak{pk}_U, \mathfrak{sk}_U)$, where $\mathfrak{sk}_U$ will be held by the party and the public key $\mathfrak{pk}_U$ will be held (in a certified form) by all other parties. It may be the case that parties have two public/private key pairs, one for public key encryption and one for public key signatures, in which case we will denote them by $(\mathfrak{pk}_U^{(e)}, \mathfrak{sk}_U^{(e)})$ and $(\mathfrak{pk}_U^{(s)}, \mathfrak{sk}_U^{(s)})$. Each party U will have a state $\mathsf{state}_U = \{\mathfrak{k}_U, \kappa_U\}$, which consists of the secret data above $\mathfrak{k}_U$ and a Boolean variable κ_U. The variable κ_U denotes whether the party is corrupted or not. At the start of the game κ_U is set to false.

For public key protocols we are also going to have a set of users $V \in \mathcal{V}$ for whom the adversary can register its own public keys. For the users in $\mathcal{V}$ we do not assume that the adversary knows the underlying private key, only that it registers public keys of its choice.

The adversary can interact with these *parties* in the following manner. It has an oracle $\mathcal{O}_U$, for each user $U \in \mathcal{U}$, to which it can pass a single command $\mathsf{corrupt}$; on receiving this command the game returns the value $\mathfrak{k}_U$ to the adversary and sets $\kappa_U \leftarrow \mathsf{true}$. Thus this oracle allows the adversary to take control of any party she desires. In addition there is an oracle $\mathcal{O}_V$, for each user $V \in \mathcal{V}$, to which it can pass the single command $(\mathsf{register}, \mathfrak{pk}_V)$ which registers $\mathfrak{pk}_V$ as the public key of the user V.

Modelling the Sessions: As well as the data associated with each party we have data associated with the *view* of a party of a session in which it is engaging. Note that this is about the view of the party and may not correspond to the truth. Each party $U \in \mathcal{U}$ may have multiple sessions

which it thinks it is having with party $U' \in \mathcal{U} \cup \mathcal{V}$; party U will index these sessions by an integer variable i. The goal of the session is to agree an ephemeral secret key $\mathfrak{ek}_{U,U',i}$, and as the key agreement protocol proceeds the session will have different states, which we shall denote by $\Sigma_{U,U',i}$. The possible states are:

- $\perp$: This is the initial value of $\Sigma_{U,U',i}$ and indicates that nothing has happened.
- accept: This indicates that party U thinks that a key has been agreed, and party U thinks it is equal to $\mathfrak{ek}_{U,U',i}$.
- reject: This indicates that party U has rejected this session of the protocol.
- revealed: This indicates that the adversary has obtained the key $\mathfrak{ek}_{U,U',i}$; see below for how she did this.

Within a session we also maintain a list of messages received and sent, a so-called transcript $\mathcal{T}_{U,U',i} = \{m_1, r_1, m_2, r_2, \ldots, m_n, r_n\}$, where m_j is the jth message received by party U in its ith session with U', and r_n is the associated response which it made. The transcript is initially set to be the empty set. Thus the session state is given by $\mathsf{session}_{U,U',i} = \{\mathfrak{ek}_{U,U',i}, \Sigma_{U,U',i}, \mathcal{T}_{U,U',i}, s_{U,U',i}\}$, where $s_{U,U',i}$ is some protocol specific state information.

To obtain information about the sessions, and to drive the protocol, the adversary has a message oracle $\mathcal{O}$ which takes four arguments, $\mathcal{O}(U, U', i, m)$, which is processed as follows:

- If $m =$ reveal and $\Sigma_{U,U',i} =$ accept then set $\Sigma_{U,U',i} \leftarrow$ revealed and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m =$ init and $\mathcal{T}_{U,U',i} = \emptyset$ then start a new protocol session with U' and let r be the initial message, set $\mathcal{T}_{U,U',i} \leftarrow \{m, r\}$ and return r to the adversary.
- Otherwise if $\mathcal{T}_{U,U',i} = \{m_1, r_1, m_2, r_2, \ldots, m_n, r_n\}$ and the message m is the $(n+1)$st input message in the protocol. The oracle computes the response r_{n+1} and adds the message and response to $\mathcal{T}_{U,U',i}$ and returns r_{n+1} to the adversary. The final response in a protocol, to signal the end, is to set $r_{n+1} \leftarrow \perp$.

Notice that messages are not sent between participants; we allow the adversary to do that. She can decide which message gets sent where, or if it even gets sent at all.

Matching Conversations: We now need to define when two such sessions, run by different participants, correspond to the same protocol execution. This is done using the concept of a matching conversation. Suppose we have two transcripts

$$\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m_2, r_2, \ldots, m_n, r_n\},$$
$$\mathcal{T}_{W,W',i'} = \{m'_1, r'_1, m'_2, r'_2, \ldots, m_k, r_k\},$$

such that

- $m'_i = r_{i-1}$ for $i \geq 1$,
- $m_i = r'_{i-1}$ for $i > 1$,
- n even: $r_n = \perp$ and $k = n - 1$,
- n odd: $r_k = \perp$ and $k = n$,
- $U = W'$ and $U' = W$.

In such a situation we say the two transcripts are matching conversations.

Winning the Game: Recall that our winning condition is going to be akin to the Real-or-Random definition for encryption security. So the game selects a hidden bit b; if $b = 0$ then the adversary will be given a random key for its challenge, otherwise it will be given a real key. So we need to decide on which key agreement protocol execution the adversary will be challenged. Instead of defining one for the adversary we let the adversary choose its own session which which it wants to be queried, subject to some constraints which we shall discuss below. This is done by means of a so-called test oracle $\mathcal{O}_{\mathsf{Test}}$.

The test oracle takes as input a tuple (U, U', i), with $U, U' \in \mathcal{U}$ and if $b = 1$ it returns $\mathfrak{ek}_{U,U',i}$, and if $b = 0$ it returns a random key of the same size as $\mathfrak{ek}_{U,U',i}$. The test oracle may only be called once by the adversary, and when making the call the adversary has to obey certain rules of the game. The restrictions on the test oracle query are for two reasons; firstly to ensure that the query makes sense, and secondly to ensure that the game is not too easy to win. For our purposes the restrictions will be as follows:

- $\Sigma_{U,U',i} = \mathsf{accept}$. Otherwise the key is either not defined, or it has been revealed and the adversary already knows it.
- There should be no revealed session (W, W', i') which has a matching conversation with (U, U', i). Since a matching conversation should correspond to the same session key, this protects against a trivial break in these circumstances.
- $\kappa_{U'} = \kappa_U = \mathsf{false}$. In other words both U and its partner U' are not corrupted.

Note that we do not assume that the test session has any matching conversations at all. It could be a conversation with the adversary.

At the end of the game (which we call the Authenticated Key Agreement game) the adversary A outputs its guess b' as to the hidden bit b, and we define the advantage of the adversary A against protocol Π in the usual way as

$$\mathrm{Adv}_{\Pi}^{\mathsf{AKA}}(A; n_P, n_S) = 2 \cdot \left| \Pr[A \text{ wins}] - \frac{1}{2} \right|,$$

where n_P upper bounds the number of participants with which A interacts and n_S upper bounds the number of sessions executable between two participants. To avoid trivial breaks of some simple protocols we also impose the restriction that after the test query on (U, U', i) is made we do not allow calls to $\mathcal{O}_U$ and $\mathcal{O}_{U'}$, i.e. at the end of the game U and U' must still be uncorrupted. If we want to model protocols with forward secrecy then we remove this restriction; thus the test session will consist of messages which were sent and received by U before the corruption of either party occured. We shall not discuss forward secrecy anymore, except to point out when protocols are not forward secure.

For a protocol to be deemed secure we need more than just that the advantage is small.

Definition 18.1 (AKA Security)**.** *An authenticated key agreement protocol* Π *is said to be secure if*

(1) *There is a matching conversation between sessions* (U, U', i) *and* (U', U, j) *and* U *and* U' *are uncorrupted, then we have* $\Sigma_{U,U',i} = \Sigma_{U',U,j} = \mathsf{accept}$ *and* $\mathfrak{ek} = \mathfrak{ek}_{U,U',i} = \mathfrak{ek}_{U',U,j}$*, and* $\mathfrak{ek}$ *is distributed uniformly at random over the desired key space.*
(2) $\mathrm{Adv}_{\Pi}^{\mathsf{AKA}}(A)$ *is "small".*

Public-Key-Encryption-Based Key Transport Example: To define a protocol within the above game framework we simply need to provide the information as to how the oracle $\mathcal{O}(U, U', i, m)$ should behave. We present a number of examples, all based on public key techniques; we leave symmetric-key-based protocols to the reader. We start with the basic key transport based on public key encryption, in particular the third version of the protocol given in Figure 18.11. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle in this case is given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then set $\mathfrak{ek}_{U,U',i} \leftarrow \mathbb{K}$, $c_U \leftarrow e^{(e)}_{\mathfrak{pk}_{U'}}(U \| \mathfrak{ek}_{U,U',i})$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U^{(s)}}(c)$, $r \leftarrow (c_U, s_U)$, $\sigma_{U,U',i} \leftarrow \mathsf{accept}$. $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m = (c_{U'}, s_{U'})$, $\mathcal{T}_{U,U',i} = \emptyset$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}^{(s)}}(s_{U'}, c_{U'}) = \mathsf{true}$ then set $m^* \leftarrow e_{\mathfrak{sk}_{U'}^{(e)}}(c)$. If $m^* = \perp$ then abort. Parse m^* as $(A \| \mathfrak{ek}_{U,U',i})$ and abort if $A \neq U'$. Finally set $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, \perp\}$, and return $\perp$ to the adversary.

- Otherwise abort.

However, earlier we said that this had a replay attack. This replay attack can now be described in our AKA model. We take the two-user case, and pass the output from the initiator to the responder twice. Thus the responder thinks they are engaging with two sessions with the initiator. We then reveal on one of these sessions and by testing the other and so we can decide is the test oracle returns a random value or not.

- Assume two parties U and U'.
- $(c_U, s_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $\mathcal{O}(U', U, 1, (c_U, s_U))$.
- $\mathcal{O}(U', U, 2, (c_U, s_U))$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', U, 2, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

An obvious fix is to remove the ability for replays, but without each party maintaining a large state this is relatively complex. As remarked above, the preferred fix would be to have each party contribute entropy to each protocol run.

Diffie–Hellman Example: We now turn to the basic Diffie–Hellman protocol from Figure 18.12. We assume we are working in a finite abelian group G of order q generated by g. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle in this case is given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathfrak{ek}_{U,U',i} \leftarrow m^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1\}$ then $\mathfrak{ek}_{U,U',i} \leftarrow m^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m, \perp\}$, and return $\perp$ to the adversary.
- Otherwise abort.

We already know this is not a secure authenticated key agreement protocol due to the man-in-the-middle attack. Indeed no public keys are even used within the protocol, so it does not have any authentication at all within it. However, to illustrate how this is captured in the AKA security model we present an attack, *within the model*. Our adversary performs the following steps:

- Assume four distinct parties U, U', K and L.
- $e_U \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $e_K \leftarrow \mathcal{O}(K, L, 1, e_U)$.
- $\mathcal{O}(U, U', 1, e_K)$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(K, L, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

Note that this attack works since the test session $(U, U', 1)$ does not have a matching conversation with *any* other session. It does "match" with the session $(K, L, 1)$ in terms of message flows, but we do not have $U = L$ and $U' = K$. This allows the adversary to reveal the key for session $(K, L, 1)$, whilst still allowing session $(U, U', 1)$ to be passed to the test oracle. Also note that the sessions $(U, U', 1)$ and $(K, L, 1)$ agree on the same key, and that neither U nor U' have been corrupted. Thus the above method means the adversary can win the AKA security game with probability one for the Diffie–Hellman protocol.

Signed Diffie–Hellman Example: We now turn to the Signed Diffie–Hellman protocol from Figure 18.13. The "code" for the $\mathcal{O}(U, U', i, m)$ oracle is now given by:

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $e_U \leftarrow g^{s_{U,U',i}}$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U}(e_U)$, $r \leftarrow (e_U, s_U)$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m = (e_{U'}, s_{U'})$ with $e_{U'} \in G$, $\mathcal{T}_{U,U',i} = \emptyset$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}}(s_{U'}, e_{U'}) = \mathsf{true}$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $e_U \leftarrow g^{s_{U,U',i}}$, $s_U \leftarrow \mathsf{Sig}_{\mathfrak{sk}_U}(e_U)$, $r \leftarrow (e_U, s_U)$, $\mathfrak{ek}_{U,U',i} \leftarrow {e_{U'}}^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$, and return r to the adversary.
- If $m = (e_{U'}, s_{U'})$ with $e_{U'} \in G$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, (e_U, s_U)\}$ and $\mathsf{Verify}_{\mathfrak{pk}_{U'}}(s_{U'}, e_{U'}) = \mathsf{true}$ then $\mathfrak{ek}_{U,U',i} \leftarrow {e_{U'}}^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, (e_U, s_U), m, \perp\}$, and return $\perp$ to the adversary.
- Otherwise abort.

Within the AKA security model we can present the following attack, which formalizes the replacement of signature attack discussed above. Our adversary performs the following steps:

- Assume three distinct parties U, U' and K.
- $\mathfrak{sk}_K \leftarrow \mathcal{O}_K(\mathsf{corrupt})$.
- $(e_U, s_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $s_K \leftarrow \mathsf{Sig}_{\mathfrak{sk}_K}(e_U)$.
- $(e_{U'}, s_{U'}) \leftarrow \mathcal{O}(U', K, 1, (e_U, s_K))$.
- $\mathcal{O}(U, U', 1, (e_{U'}, s_{U'}))$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', K, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

In this execution only party K is corrupted, and in particular neither U nor U' have been corrupted. However, party U thinks it is talking to party U', whereas party U' thinks it is talking to party K; yet in the sessions executed by U and U' they agree on the same ephemeral key. Hence, the above attack is valid within the AKA model and succeeds with probability one.

Blake-Wilson–Menezes Example: It turns out that the Blake-Wilson–Menezes protocol also has an attack on it, which makes use of the fact that we allow the adversary to register public keys of its own choosing. We describe the attack via the following pseudo-code, assuming the underlying group is of prime order q.

- Assume two legitimate parties U and U'.
- $t \leftarrow (\mathbb{Z}/q\mathbb{Z})^*$.
- $\mathfrak{pk}_V \leftarrow {\mathfrak{pk}_U}^t$.
- $\mathcal{O}_V(\mathsf{register}, \mathfrak{pk}_V)$.
- $(e_U) \leftarrow \mathcal{O}(U, U', 1, \mathsf{init})$.
- $(e_{U'}) \leftarrow \mathcal{O}(U', V, 1, e_U)$.
- $\mathcal{O}(U, U', 1, e_{U'}^t)$.
- $\mathfrak{k}' \leftarrow \mathcal{O}(U', V, 1, \mathsf{reveal})$.
- $\mathfrak{k} \leftarrow \mathcal{O}_{\mathsf{Test}}(U, U', 1)$.
- If $\mathfrak{k} = \mathfrak{k}'$ then return $b' = 1$.
- Return $b' = 0$.

Notice that U thinks it is talking to U', whereas U' thinks it is talking to V. Let x be the secret ephemeral key chosen by U in the above execution, and y is the secret ephemeral key chosen by U'. Then the secret key for the (test) session U is engaged in is equal to

$$\mathfrak{k} \leftarrow H({\mathfrak{pk}_{U'}}^x, {\mathfrak{ek}_{U'}}^{t \cdot s_U}) = H(g^{s_{U'} \cdot x}, g^{t \cdot y \cdot s_U}),$$

whilst the secret key for the (revealed) session U' is engaged in is equal to

$$\mathfrak{k}' \leftarrow H({\mathfrak{ek}_V}^{s_{U'}}, {\mathfrak{pk}_V}^{y}) = H(g^{x \cdot s_{U'}}, g^{t \cdot s_U \cdot y}).$$

So the two keys are identical, and yet the session executed by U' is not a matching session with that executed by U.

Modified Blake-Wilson–Menezes Example: The problem with the previous protocol is that the key is derived does not depend on whether the sessions involved have matching conversations. We therefore modify the Blake-Wilson–Menezes protocol in the following way, and we are then able to prove the protocol is secure. Again we assume public/private key pairs are given by $\mathfrak{pk}_A = g^a$ for Alice and $\mathfrak{pk}_B = g^b$ for Bob, and that the message flows are still defined by

$$A \longrightarrow B : \mathfrak{ek}_A = g^x,$$
$$B \longrightarrow A : \mathfrak{ek}_B = g^y.$$

The key difference is in how the shared secret key k is derived; we also include the transcript of the protocol within the key derivation. Alice derives it via the equation

$$k \leftarrow H({\mathfrak{pk}_B}^{x}, {\mathfrak{ek}_B}^{a}, \mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B) = H(g^{b \cdot x}, g^{y \cdot a}, g^a, g^b, g^x, g^y).$$

While Bob derives the same key using the equation

$$k \leftarrow H({\mathfrak{ek}_A}^{b}, {\mathfrak{pk}_A}^{y}, \mathfrak{pk}_A, \mathfrak{pk}_B, \mathfrak{ek}_A, \mathfrak{ek}_B) = H(g^{x \cdot b}, g^{a \cdot y}, g^a, g^b, g^x, g^y).$$

In terms of "code" for our $\mathcal{O}(U, U', i, m)$ oracle we have that the protocol is defined by

- If $m = \mathsf{reveal}$ and $\Sigma_{U,U',i} = \mathsf{accept}$ then set $\Sigma_{U,U',i} \leftarrow \mathsf{revealed}$ and return $\mathfrak{ek}_{U,U',i}$ to the adversary.
- If $m = \mathsf{init}$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r\}$, and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \emptyset$ then $s_{U,U',i} \leftarrow \mathbb{Z}/q\mathbb{Z}$, $r \leftarrow g^{s_{U,U',i}}$, $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{m, r\}$,
$$\mathfrak{ek}_{U,U',i} \leftarrow H(m^{\mathfrak{sk}_U}, {\mathfrak{pk}_{U'}}^{s_{U,U',i}}, \mathfrak{pk}_{U'}, \mathfrak{pk}_U, m, r),$$
and return r to the adversary.
- If $m \in G$ and $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1\}$ then $\Sigma_{U,U',i} \leftarrow \mathsf{accept}$, $\mathcal{T}_{U,U',i} = \{\mathsf{init}, r_1, m, \perp\}$,
$$\mathfrak{ek}_{U,U',i} \leftarrow H({\mathfrak{pk}_{U'}}^{s_{U,U',i}}, m^{s_U}, \mathfrak{pk}_U, \mathfrak{pk}_{U'}, r_1, m),$$
and return $\perp$ to the adversary.
- Otherwise abort.

We can now prove that this protocol is secure assuming the Gap Diffie–Hellman problem is hard.

Theorem 18.2. *Let A denote an adversary against the* AKA *security of the modified Blake-Wilson–Menezes protocol, operating with n_P legitimate users each of whom may have up to n_S sessions, where H is modelled as a random oracle. Then there is an adversary B against the Gap Diffie–Hellman problem in the group G such that*

$$\mathrm{Adv}^{\mathsf{AKA}}_{\Pi}(A; n_P, n_S) \leq n_P^2 \cdot n_S \cdot \mathrm{Adv}^{\mathsf{Gap\text{-}DHP}}_{G}(B).$$

From the theorem it is easy to deduce that the modifed Blake-Wilson–Menezes protocol is secure, assuming the Gap Diffie–Hellman problem is hard.

PROOF. We assume we are given an algorithm A against the AKA security of the protocol and we want to create the algorithm B against the Gap Diffie–Hellman problem. Algorithm B will maintain a hash list H-List consisting of elements in $G^6 \times \{0,1\}^k$, representing the calls to H on sextuples of elements in G and the corresponding outputs.

Algorithm B has as input g^a and g^y and is asked to compute $g^{a \cdot y}$, given access to an oracle $\mathcal{O}_{\mathsf{DDH}}$ which checks tuples for being Diffie–Hellman tuples. Algorithm B sets up n_P public/private

keys for algorithm A as in the legitimate game, but picks one user $U^* \in \mathcal{U}$ and sets its public key to be g^a. Algorithm B also picks a session identifier $i^* \in \{1, \ldots, n_S\}$, and another identity $W^* \in \mathcal{U}$. We let b denote the private key of entity W^*; notice that this is known to algorithm B, and hence is marked in blue.

Algorithm B then calls algorithm A and responds to its oracle queries as in the real game except for the following cases:

- If $\mathcal{O}_{U^*}(\mathsf{corrupt})$ is called then abort.
- If $\mathcal{O}(U^*, W^*, i^*, \mathsf{reveal})$ is called then abort.
- If $\mathcal{O}(W^*, U^*, i^*, m)$ is called then respond with g^x.
- If $\mathcal{O}_{\mathsf{Test}}(U, W, i)$ is called with $U \neq U^*$, $W \neq W^*$ or $i \neq i^*$ then abort. Otherwise respond with a random value in $\{0,1\}^k$.

The only problem is in maintaining consistency between any reveal queries made by A and any calls made by A to the hash function H. In other words we need to maintain consistency of the H-List in both the reveal and hash function queries made by A. However, such consistency can be maintained in the same manner as in Theorem 16.9 using the $\mathcal{O}_{\mathsf{DDH}}$ oracle to which algorithm B has access[2].

Note that the probability that B aborts is $1/(n_P^2 \cdot n_S)$. If the algorithm A is able to win the AKA game with non-negligible probability then A must make a *hash* query on the critical sextuple. As we are using the modified Blake-Wilson–Menezes protocol, no reveal query made by A will result in the same input to the hash function as the critical query. Since no other reveal query will have the same transcript etc. Thus to have any advantage A must make the critical call to the hash function.

We now examine what this sextuple will be; we let m denote the ephemeral public key passed to W^* in the test session and let $x = \mathrm{dlog}_g(m)$; note that B does not know x so we mark this value in red. If U^* was an initiator oracle then the sextuple is

$$(\mathfrak{pk}_{W^*}{}^x, \mathfrak{ek}_{W^*}{}^a, \mathfrak{pk}_{U^*}, \mathfrak{pk}_{W^*}, \mathfrak{ek}_{U^*}, \mathfrak{ek}_{W^*}) = (\mathfrak{pk}_{W^*}{}^x, g^{y \cdot a}, g^a, \mathfrak{pk}_{W^*}, m, g^y)\,.$$

Thus in this case the algorithm B simply looks for the critical query on the H-List using its $\mathcal{O}_{\mathsf{DDH}}$ oracle and outputs the second value of the tuple as the answer to the Gap Diffie–Hellman problem. If U^* was the responder oracle, then a similar method can be applied using the first value of the tuple, as the critical sextuple is given by

$$(\mathfrak{ek}_{W^*}{}^a, \mathfrak{pk}_{W^*}{}^x, \mathfrak{pk}_{W^*}, \mathfrak{pk}_{U^*}, \mathfrak{ek}_{W^*}, \mathfrak{ek}_{U^*}) = (g^{y \cdot a}, \mathfrak{pk}_{W^*}{}^x, \mathfrak{pk}_{W^*}, g^a, g^y, m)\,.$$

□

A similar proof can be made for the MQV protocol, which due to space we do not cover here.

Chapter Summary

- Digital certificates allow us to bind a public key to some other information, such as an identity. This binding of key with identity allows us to solve the problem of how to distribute authentic public keys.
- Various PKI systems have been proposed, all of which have problems and benefits associated with them.
- Implicit certificates aim to reduce the bandwidth requirements of standard certificates, however they come with a number of drawbacks.

[2]The details are tedious and are left to the reader.

- A number of key agreement protocols exist based on a trusted third party and symmetric encryption algorithms. These protocols require long-term keys to have been already established with the TTP; they may also require some form of clock synchronization.
- Diffie–Hellman key exchange can be used by two parties to agree on a secret key over an insecure channel. However, Diffie–Hellman is susceptible to a man-in-the-middle attack and so requires some form of authentication of the communicating parties.
- To obtain authentication in the Diffie–Hellman protocol, various different options exist, of which we discussed the STS protocol, the Blake-Wilson–Menezes protocol and the MQV protocol.
- Various formal logics exist to analyse such protocols. The most influential of these has been the BAN logic. These logics help to identify explicit assumptions and problems associated with each protocol; they can identify attacks but usually do not provide security proofs.
- For a computational proof of security one needs to define an elaborate security model. Such models capture a multitude of attacks; resulting in the most simple protocols being deemed insecure.

Further Reading

The paper by Burrows, Abadi and Needham is a very readable introduction to the BAN logic and a number of key agreement protocols based on static symmetric keys; much of our treatment of this subject is based on this paper. Our treatment of security models for key agreement is based on work started in the paper by Bellare and Rogaway. Our proof of security of the modified Blake-Wilson–Menezes protocol is based on the paper by Kudla and Paterson.

M. Bellare and P. Rogaway. Entity authentication and key distribution. Advances in Cryptology – Crypto 1993, LNCS 773, 232–249, Springer, 1994.

M. Burrows, M. Abadi and R. Needham. *A Logic of Authentication.* Digital Equipment Corporation, SRC Research Report 39, 1990.

C. Kudla and K.G. Paterson. *Modular security proofs for key agreement protocols.* Advances in Cryptology – Asiacrypt 2005, LNCS 3788, 549–565, Springer, 2005.

Part 4

Advanced Protocols

Encryption, hash functions, MACs and signatures are only the most basic of cryptographic constructions and protocols. We usually think of them as being carried out between a sender and a receiver who have the same security goals. For example, in encryption both the sender and the receiver probably wish to keep the message secret from an adversary. In other words the adversary is assumed to be someone else.

In this section we shall detail a number of more advanced protocols. These are mainly protocols between two or more people in which the security goals of the different parties could be conflicting, or different. For example in an electronic election voters want their votes to be secret, yet all parties want to know that all votes have been counted, and all parties want to ensure against a bad voter casting too many votes or trying to work out how someone else has voted. Hence, the adversaries are also the parties in the protocol, not necessarily external entities.

First we focus on secret sharing schemes, which allow a party to share a secret amongst a number of partners. This has important applications in splitting of secrets into parts which can then be used in distributed protocols. Then we turn to commitment schemes and oblivious transfer. These are two types of basic protocols between two parties, in which the parties are assumed to be mutually untrusting, i.e. the adversary is the person with whom you are performing the protocol. We then turn to the concept of zero-knowledge proofs. In this chapter we also examine a simple electronic voting scheme. Finally we look at the subject of secure multi-party computation, which provides an interesting application of many of our preceding algorithms.

CHAPTER 19

Secret Sharing Schemes

Chapter Goals

- To introduce the notion of secret sharing schemes.
- To give some simple examples of general access structures.
- To present Shamir's scheme, including how to recover the secret in the presence of active adversaries.
- To show the link between Shamir's secret sharing and Reed–Solomon codes.
- As an application we show how secret sharing can provide more security for a certificate authority, via distributed RSA signature generation.

19.1. Access Structures

Suppose you have a secret s which you wish to share amongst a set $\mathcal{P}$ of n parties. You would like certain subsets of the n parties to recover the secret but not others. The classic scenario might be that s is a nuclear launch code and you have four people, the president, the vice-president, the secretary of state and a general in a missile silo. You do not want the general to be able to launch the missile without the president agreeing, but to maintain deterrence you would like, in the case that the president has been eliminated, that the vice-president, the secretary of state and the general can agree to launch the missile. If we label the four parties as P, V, S and G, for president, vice-president, secretary of state and general, then we would like the following sets of people to be able to launch the missile

$$\{P, G\} \text{ and } \{V, S, G\},$$

but no smaller sets. It is this problem which secret sharing is designed to deal with, however the applications are more widespread than might at first appear.

To each party we distribute some information called a *share*. For a party A we will let s_A denote the secret share which they hold. In the example above there are four such shares: s_P, s_V, s_S and s_G. Then if the required parties come together we would like an algorithm which combines their relevant shares into the secret s. But if the wrong set of parties come together they should learn no information about s.

Before introducing schemes to perform secret sharing we first need to introduce the notion of an access structure. Any subset of parties who can recover the secret will be called a *qualifying set*, whilst the set of all qualifying sets will be called an *access structure*. So in the example above we have that the two sets

$$\{P, G\} \text{ and } \{V, S, G\}$$

are qualifying sets. However, clearly any set containing such a qualifying set is also a qualifying set. Thus

$$\{P, G, V\},\ \{P, G, S\} \text{ and } \{P, V, G, S\}$$

are also qualifying sets. Hence, there are five sets in the access structure. For any set in the access structure, if we have the set of shares for that set we would like to be able to reconstruct the secret.

Definition 19.1. *A monotone structure on a set $\mathcal{P}$ is a collection Γ of subsets of $\mathcal{P}$ such that*

- $\mathcal{P} \in \Gamma$.
- *If there is a set $A \in \Gamma$ and a set B such that $A \subset B \subset \mathcal{P}$ then $B \in \Gamma$.*

Thus in the above example the access structure is monotone. This is a property which will hold for all access structures of all secret sharing schemes. For a monotone structure we note that the sets in Γ come in chains, $A \subset B \subset C \subset \mathcal{P}$. We shall call the sets which form the start of a chain the *minimal qualifying sets*. The set of all such minimal qualifying sets for an access structure Γ we shall denote by $m(\Gamma)$. We can now give a very informal definition of what we mean by a secret sharing scheme:

Definition 19.2. *A secret sharing scheme for a monotone access structure Γ over a set of parties $\mathcal{P}$ with respect to a space of secrets S is a pair of algorithms called* `Share` *and* `Recombine` *with the following properties:*

- `Share`(s, Γ) *takes a secret $s \in S$ and a monotone access structure and determines a value s_A for every $A \in \mathcal{P}$. The value s_A is called A's share of the secret.*
- `Recombine`(H) *takes a set $H_{\mathcal{O}}$ of shares for some subset $\mathcal{O}$ of $\mathcal{P}$, i.e.*

$$H_{\mathcal{O}} = \{s_O : O \in \mathcal{O}\}.$$

If $\mathcal{O} \in \Gamma$ then this should return the secret s, otherwise it should return nothing.

A secret sharing scheme is considered to be secure if no infinitely powerful adversary can learn anything about the underlying secret without having access to the shares of a qualifying set. Actually such schemes are said to be information-theoretically secure, but since most secret sharing schemes in the literature are information-theoretically secure we shall just call such schemes *secure*.

In this chapter we will consider two running examples of monotone access structures, so as to illustrate the schemes. Both will be on sets of four elements: The first is from the example above where we have $\mathcal{P} = \{P, V, S, G\}$ and

$$\Gamma = \{\{P, G\}, \{V, S, G\}, \{P, G, V\}, \{P, G, S\}, \{P, V, G, S\}\}.$$

The set of minimal qualifying sets is given by

$$m(\Gamma) = \{\{P, G\}, \{V, S, G\}\}.$$

The second example we shall define over the set of parties $\mathcal{P} = \{A, B, C, D\}$, with access structure

$$\begin{aligned}\Gamma = \{&\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\},\\ &\{A, B, C\}, \{A, B, D\}, \{B, C, D\}, \{A, B, C, D\}\}.\end{aligned}$$

The set of minimal qualifying sets is given by

$$m(\Gamma) = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}.$$

This last access structure is interesting because it represents a common form of threshold access structure. Notice that, in this access structure, we require that any two out of the four parties should be able to recover the secret. We call such a scheme a 2-out-of-4 threshold access structure.

One way of looking at such access structures is via a Boolean formulae. Consider the set $m(\Gamma)$ of minimal qualifying sets and define the formula:

$$\bigvee_{\mathcal{O} \in m(\Gamma)} \left(\bigwedge_{O \in \mathcal{O}} O \right).$$

So in our first example above the formula becomes

$$(P \wedge G) \vee (V \wedge S \wedge G).$$

Reading this formula out, with $\wedge$ being "and", and $\vee$ being "or", we see that we can reconstruct the secret if we have access to the secret shares of

$$\text{(P and G) or (V and S and G).}$$

Notice how the formula is in disjunctive normal form (DNF), i.e. an "or" of a set of "and" clauses. We shall use this representation below to construct a secret sharing scheme for any access structure.

19.2. General Secret Sharing

We now turn to two methods for constructing secret sharing schemes for arbitrary monotone access structures. They are highly inefficient for all but the simplest access structures but they do show that one can cope with an arbitrary access structure. We assume that the space of secrets S is essentially the set of bit strings of length n bits. In both examples we let $s \in S$ denote the secret which we are trying to share.

19.2.1. Ito–Nishizeki–Saito Secret Sharing: Our first secret sharing scheme makes use of the DNF Boolean formula we presented above. In some sense every "or" gets converted into a concatenation operation and every "and" gets converted into a $\oplus$ operation. This can at first sight seem slightly counterintuitive, since usually we associate "and" with multiplication and "or" with addition.

The sharing algorithm works as follows. For every minimal qualifying set $\mathcal{O} \in m(\Gamma)$, we generate shares $s_i \in S$, for $1 \leq i \leq l$, at random, where $l = |\mathcal{O}|$ such that $s_1 \oplus \cdots \oplus s_l = s$. Then a party A is given a share s_i if A occurs at position i in the set $\mathcal{O}$.

Example: Recall we have the formula

$$\text{(P and G) or (V and S and G).}$$

We generate five elements s_i from S such that

$$\begin{aligned} s &= s_1 \oplus s_2, \\ &= s_3 \oplus s_4 \oplus s_5. \end{aligned}$$

The four shares are then defined to be:

$$\begin{aligned} s_P &\leftarrow s_1, \\ s_V &\leftarrow s_3, \\ s_S &\leftarrow s_4, \\ s_G &\leftarrow s_2 \| s_5. \end{aligned}$$

You should check that, given this sharing, any qualifying set can recover the secret, and only the qualifying sets can recover the secret. Notice that party G needs to hold two times more data than the size of the secret. Thus this scheme in this case is not efficient. Ideally we would like the parties to only hold the equivalent of n bits of information each so as to recover a secret of n bits.

Example: Now our formula is given by

$$\text{(A and B) or (A and C) or (A and D) or (B and C) or (B and D) or (C and D).}$$

We now generate twelve elements s_i from S, one for each of the distinct terms in the formula above, such that

$$\begin{aligned} s &= s_1 \oplus s_2, \\ &= s_3 \oplus s_4, \\ &= s_5 \oplus s_6, \\ &= s_7 \oplus s_8, \\ &= s_9 \oplus s_{10}, \\ &= s_{11} \oplus s_{12}. \end{aligned}$$

The four shares are then defined to be:

$$\begin{aligned} s_A &\leftarrow s_1 \| s_3 \| s_5, \\ s_B &\leftarrow s_2 \| s_7 \| s_9, \\ s_C &\leftarrow s_4 \| s_8 \| s_{11}, \\ s_D &\leftarrow s_6 \| s_{10} \| s_{12}. \end{aligned}$$

You should again check that, given this sharing, any qualifying set and only the qualifying sets can recover the secret. We see that in this case every share contains three times more information than the underlying secret.

19.2.2. Replicated Secret Sharing: The above is not the only scheme for general access structures. Here we present another one, called the replicated secret sharing scheme. In this scheme we first create the sets of all maximal non-qualifying sets; these are the sets of all parties such that if you add a single new party to each set you will obtain a qualifying set. If we label these sets $A_1, \ldots, A_t$, we then form their set-theoretic complements, i.e. $B_i = \mathcal{P} \setminus A_i$. Then a set of secret shares s_i is then generated, one for each set B_i, so that

$$s = s_1 \oplus \cdots \oplus s_t.$$

A party is given the share s_i if it is contained in the set B_i.

Example: The sets of maximal non-qualifying sets for our first example are

$$A_1 = \{P, V, S\},\ A_2 = \{V, G\} \text{ and } A_3 = \{S, G\}.$$

Forming their complements we obtain the sets

$$B_1 = \{G\},\ B_2 = \{P, S\} \text{ and } B_3 = \{P, V\}.$$

We generate three shares s_1, s_2 and s_3 such that $s = s_1 \oplus s_2 \oplus s_3$ and then define the shares as

$$\begin{aligned} s_P &\leftarrow s_2 \| s_3, \\ s_V &\leftarrow s_3, \\ s_S &\leftarrow s_2, \\ s_G &\leftarrow s_1. \end{aligned}$$

Again we can check that only the qualifying sets can recover the secret.

Example: For the 2-out-of-4 threshold access structure we obtain the following maximal non-qualifying sets

$$A_1 = \{A\},\ A_2 = \{B\},\ A_3 = \{C\} \text{ and } A_4 = \{D\}.$$

On forming their complements we obtain

$$B_1 = \{B, C, D\},\ B_2 = \{A, C, D\},\ B_3 = \{A, B, D\} \text{ and } B_4 = \{A, B, C\}.$$

We form the four shares such that $s = s_1 \oplus s_2 \oplus s_3 \oplus s_4$ and set

$$\begin{aligned} s_A &\leftarrow s_2 \| s_3 \| s_4, \\ s_B &\leftarrow s_1 \| s_3 \| s_4, \\ s_C &\leftarrow s_1 \| s_2 \| s_4, \\ s_D &\leftarrow s_1 \| s_2 \| s_3. \end{aligned}$$

Whilst the above two constructions provide a mechanism to construct a secret sharing scheme for any monotone access structure, they appear to be very inefficient. In particular for the threshold access structure they are especially bad, especially as the number of parties increases. In the rest of this chapter we will examine a very efficient mechanism for threshold secret sharing due to Shamir, called Shamir secret sharing. This secret sharing scheme is itself based on the ideas behind certain error-correcting codes, called Reed–Solomon codes. So we will first have a little digression into coding theory.

19.3. Reed–Solomon Codes

An error-correcting code is a mechanism to transmit data from A to B such that any errors which occur during transmission, for example due to noise, can be corrected. They are found in many areas of electronics: they are the thing which makes your CD/DVD resistant to minor scratches; they make sure that RAM chips preserve your data correctly; they are used for communication between Earth and satellites or deep space probes.

A simpler problem is that of error detection. Here one is only interested in whether the data have been altered or not. A particularly important distinction to make between the area of coding theory and cryptography is that in coding theory one can select simpler mechanisms to detect errors. This is because in coding theory the assumption is that the errors are introduced by random noise, whereas in cryptography any errors are thought to be actively inserted by an adversary. Thus in coding theory error detection mechanisms can be very simple, whereas in cryptography we have to resort to complex mechanisms such as MACs and digital signatures.

Error correction on the other hand is not only interested in detecting errors, it also wants to correct those errors. Clearly one cannot correct all errors, but it would be nice to correct a certain number. A classic way of forming error-correcting codes is via Reed–Solomon codes. In coding theory such codes are usually presented over a finite field of characteristic two. However, we are interested in the general case and so we will be using a code over $\mathbb{F}_q$, for a prime power q. Each code-word is a vector over $\mathbb{F}_q$, with the vector length being called the length of the code-word. The set of all valid code-words forms the code, there is a mapping from the set of valid code-words to the data one wants to transmit. The invalid code-words, i.e. vectors which are not in the code, are used to perform error detection and correction.

To define a Reed–Solomon code we also require two integer parameters, n and t. The value n defines the length of each code-word, whilst the number t is related to the number of errors we can correct; indeed we will be able to correct $(n-t-1)/2$ errors. We also define a set $\mathbb{X} \subset \mathbb{F}_q$ of size n. If the characteristic of $\mathbb{F}_q$ is larger than n then we can select $\mathbb{X} = \{1, 2, \ldots, n\}$, although any subset of $\mathbb{F}_q$ will do. For our application to Shamir secret sharing later on we will assume that $0 \notin \mathbb{X}$.

Consider the set of polynomials of degree less than or equal to t over the field $\mathbb{F}_q$.

$$\mathbb{P} = \left\{ f_0 + f_1 \cdot X + \ldots + f_t \cdot X^t : f_i \in \mathbb{F}_q \right\}.$$

The set $\mathbb{P}$ represents the number of code-words in our code, i.e. the number of different data items which we can transmit in any given block, and hence this number is equal to q^{t+1}. To transmit some data, in a set of size $|\mathbb{P}|$, we first encode it as an element of $\mathbb{P}$ and then we translate it into a

code-word. To create the actual code-word we evaluate the polynomial at all elements in $\mathbb{X}$. Hence, the set of *actual* code-words is given by

$$\mathcal{C} = \{(f(x_1), \ldots, f(x_n)) : f \in \mathbb{P}, x_i \in S\}.$$

The size (in bits) of a code-word is then $n \cdot \log_2 q$. So we require $n \cdot \log_2 q$ bits to represent $(t+1) \cdot \log_2 q$ bits of information.

Example: As an example consider the Reed–Solomon code with parameters $q = 101$, $n = 7$, $t = 2$ and $\mathbb{X} = \{1, 2, 3, 4, 5, 6, 7\}$. Suppose our "data", which we represent by an element of $\mathbb{P}$, is given by the polynomial

$$f = 20 + 57 \cdot X + 68 \cdot X^2.$$

To transmit this data we compute $f(i) \pmod q$ for $i = 1, \ldots, 7$, to obtain the code-word

$$c = (44, 2, 96, 23, 86, 83, 14).$$

This code-word can now be transmitted or stored.

19.3.1. Data Recovery: At some point the code-word will need to be converted back into the data. In other words we have to recover the polynomial in $\mathbb{P}$ from the set of points at which it was evaluated, i.e. the vector of values in $\mathcal{C}$. We will first deal with the simple case and assume that no errors have occurred. The receiver is given the data $c = (c_1, \ldots, c_n)$ but has no idea as to the underlying polynomial f. Thus from the receiver's perspective he wishes to find the f_i such that

$$f = \sum_{j=0}^{t} f_j \cdot X^j.$$

It is well known, from high school, that a polynomial of degree at most t is determined completely by its values at $t+1$ points. So as long as $t < n$ we can recover f when no errors occur; the question is how?

First note that the receiver can generate n linear equations via

$$c_i = f(x_i) \text{ for } x_i \in \mathbb{X}.$$

In other words he has the system of equations:

$$\begin{aligned} c_1 &= f_0 + f_1 \cdot x_1 + \cdots + f_t \cdot x_1^t, \\ &\vdots \qquad\qquad \vdots \\ c_n &= f_0 + f_1 \cdot x_n + \cdots + f_t \cdot x_n^t. \end{aligned}$$

So by solving this system of equations over the field $\mathbb{F}_q$ we can recover the polynomial and hence the data.

Actually the polynomial f can be recovered without solving the linear system, via the use of Lagrange interpolation. Suppose we first compute the polynomials

$$\delta_i(X) \leftarrow \prod_{x_j \in \mathbb{X}, j \neq i} \frac{X - x_j}{x_i - x_j}, \ 1 \leq i \leq n.$$

Note that we have the following properties, for all i,

- $\delta_i(x_i) = 1$.
- $\delta_i(x_j) = 0$, if $i \neq j$.
- $\deg \delta_i(X) = n - 1$.

Lagrange interpolation takes the values c_i and computes

$$f(X) \leftarrow \sum_{i=1}^{n} c_i \cdot \delta_i(X).$$

The three properties above of the polynomials $\delta_i(X)$ translate into the following facts about $f(X)$:

- $f(x_i) = c_i$ for all i.
- $\deg f(X) \leq n - 1$.

Hence, Lagrange interpolation finds the unique polynomial which interpolates the n elements in the code-word.

19.3.2. Error Detection: We see that by using Lagrange interpolation on the code-word we will recover a polynomial of degree t when there are no errors, but when there are errors in the received code-word we are unlikely to obtain a valid polynomial, i.e. an element of $\mathbb{P}$. Hence, we instantly have an error-detection algorithm: we apply the method above to recover the polynomial, assuming it is a valid code-word, then if the resulting polynomial has degree greater than t we can conclude that an error has occured.

Returning to our example parameters above, suppose the following code-word was received

$$c = (44, 2, 25, 23, 86, 83, 14).$$

In other words it is equal to the sent code-word except in the third position where a 96 has been replaced by a 25. We compute once and for all the polynomials, modulo $q = 101$,

$$\begin{aligned}
\delta_1(X) &= 70 \cdot X^6 + 29 \cdot X^5 + 46 \cdot X^4 + 4 \cdot X^3 + 43 \cdot X^2 + 4 \cdot X + 7,\\
\delta_2(X) &= 85 \cdot X^6 + 12 \cdot X^5 + 23 \cdot X^4 + 96 \cdot X^3 + 59 \cdot X^2 + 49 \cdot X + 80,\\
\delta_3(X) &= 40 \cdot X^6 + 10 \cdot X^5 + 83 \cdot X^4 + 23 \cdot X^3 + 48 \cdot X^2 + 64 \cdot X + 35,\\
\delta_4(X) &= 14 \cdot X^6 + 68 \cdot X^5 + 33 \cdot X^4 + 63 \cdot X^3 + 78 \cdot X^2 + 82 \cdot X + 66,\\
\delta_5(X) &= 40 \cdot X^6 + 90 \cdot X^5 + 99 \cdot X^4 + 67 \cdot X^3 + 11 \cdot X^2 + 76 \cdot X + 21,\\
\delta_6(X) &= 85 \cdot X^6 + 49 \cdot X^5 + 91 \cdot X^4 + 91 \cdot X^3 + 9 \cdot X^2 + 86 \cdot X + 94,\\
\delta_7(X) &= 70 \cdot X^6 + 45 \cdot X^5 + 29 \cdot X^4 + 60 \cdot X^3 + 55 \cdot X^2 + 43 \cdot X + 1.
\end{aligned}$$

The receiver now tries to recover the sent polynomial given the data he has received. He obtains

$$\begin{aligned}
f(X) &\leftarrow 44 \cdot \delta_1(X) + \cdots + 14 \cdot \delta_7(X)\\
&= 60 + 58 \cdot X + 94 \cdot X^2 + 84 \cdot X^3 + 66 \cdot X^4 + 98 \cdot X^5 + 89 \cdot X^6.
\end{aligned}$$

But this is a polynomial of degree six and not of degree $t = 2$. Hence, the receiver knows that there is at least one error in the code word that he has received. He just does not know which position is in error, nor what its actual value should be.

19.3.3. Error Correction: The intuition behind error correction is the following. Consider a polynomial of degree three over the reals evaluated at seven points, such as that in Figure 19.1. Clearly there is only one cubic curve which interpolates all of the points, since we have specified seven of them and we only need four such points to define a cubic curve. Now suppose one of these evaluations is given in error, for example the point at $x = 3$, as in Figure 19.2. We see that we still have six points on the cubic curve, and so there is a unique cubic curve passing through these six valid points. However, suppose we took a different set of six points, i.e. five valid ones and one incorrect one. It is then highly likely that the curve which goes through the second set of six points would not be cubic. In other words because we have far more valid points than we need to determine the cubic curve, we are able to recover it.

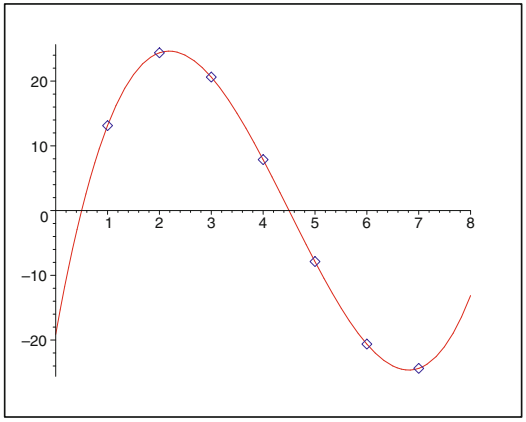


FIGURE 19.1. Cubic function evaluated at seven points

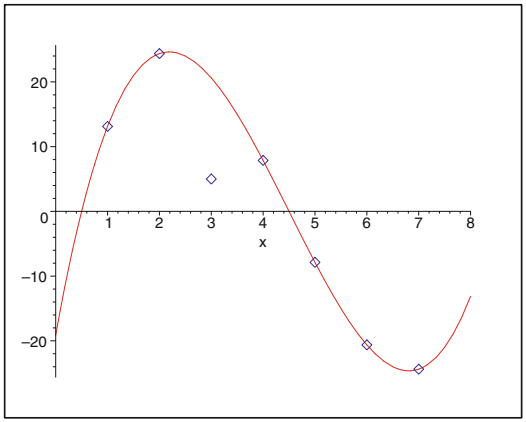


FIGURE 19.2. Cubic function going through six points and one error point

We return to the general case of a polynomial of degree t evaluated at n points. Suppose we know that there are at most e errors in our code-word. We then have the following (very inefficient) method for polynomial reconstruction.

- We produce the list of all subsets $\mathbb{X}$ of the n points with $n - e$ members.
- We then try to recover a polynomial of degree t. If we are successful then there is a good probability that the subset $\mathbb{X}$ is the valid set; if we are unsuccessful then we know that $\mathbb{X}$ contains an element which is in error.

This is clearly a silly algorithm to error correct a Reed–Solomon code. The total number of subsets we may have to take is given by

$$^{n}C_{n-e} = \frac{n!}{e! \cdot (n-e)!}.$$

But despite this we will still analyse this algorithm a bit more: To be able to recover a polynomial of degree t we must have that $t < n - e$, i.e. we must have more valid elements than there are coefficients to determine. Suppose we not only have e errors but we also have s "erasures", i.e. positions for which we do not even receive the value. Note that erasures are slightly better for the receiver than errors, since with an erasure the receiver knows the position of the erasure, whereas

with an error they do not. To recover a polynomial of degree t we will require

$$t < n - e - s.$$

But we could recover many such polynomials, for example if $t = n - e - s - 1$ then all such sets $\mathbb{X}$ of $n - e - s$ will result in a polynomial of degree at most t. To obtain a *unique* polynomial from the above method we will need to make sure we do not have too many errors/erasures.

It can be shown that if we can obtain at least $t + 2 \cdot e$ points then we can recover a unique polynomial of degree t which passes through $n-s-e$ points of the set of $n-s$ points. This gives the important equation that an error correction for Reed–Solomon codes can be performed uniquely provided

$$n > t + 2 \cdot e + s.$$

The only problem left is how to perform this error correction *efficiently*.

19.3.4. The Berlekamp–Welch Algorithm: We now present an efficient method to perform error correction for Reed–Solomon codes called the Berlekamp–Welch algorithm. The idea is to interpolate a polynomial in two variables through the points which we are given. Suppose we are given a code word with s missing values, and the number of errors is bounded by

$$e < t < \frac{n-s}{3}.$$

This means we are actually given $n - s$ supposed values of $y_i = f(x_i)$. We know the pairs (x_i, y_i) and we know that at most e of them are wrong, in that they do not come from evaluating the hidden polynomial $f(X)$. The goal of error correction is to try to recover this hidden polynomial.

We consider the bivariate polynomial

$$Q(X, Y) = f_0(X) - f_1(X) \cdot Y,$$

where f_0 (resp. f_1) is a polynomial of degree at most $2 \cdot t$ (resp. t). We impose the condition that $f_1(0) = 1$. We treat the coefficients of the f_i as variables which we want to determine. Due to the bounds on the degrees of the two polynomials, and the extra condition of $f_1(0) = 1$, we see that the number of variables we have is

$$v = (2 \cdot t + 1) + (t + 1) - 1 = 3 \cdot t + 1.$$

We would like the bivariate polynomial $Q(X, Y)$ to interpolate our points (x_i, y_i). By substituting in the values of x_i and y_i we obtain a linear equation in terms of the unknown coefficients of the polynomials f_i. Since we have $n - s$ such points, the number of linear equations we obtain is $n - s$. After determining f_0 and f_1 we then compute

$$f \leftarrow \frac{f_0}{f_1}.$$

To see that this results in the correct answer, consider the single polynomial in one variable

$$P(X) = Q(X, f(X))$$

where $f(X)$ is the polynomial we are trying to determine. We have $\deg P(X) \leq 2 \cdot t$. The polynomial $P(X)$ clearly has at least $n - s - e$ zeros, i.e. the number of valid pairs. So the number of zeros is at least

$$n - s - e > n - e - t > 3 \cdot t - t = 2 \cdot t,$$

since $e < t < \frac{n-s}{3}$. Thus $P(X)$ has more zeros than its degree, and it must hence be the zero polynomial. Hence,

$$f_0 - f_1 \cdot f = 0$$

and so $f = f_0/f_1$ since $f_1 \neq 0$.

Example: Again consider our previous example. We have received the invalid code-word

$$c = (44, 2, 25, 23, 86, 83, 14).$$

We know that the underlying code is for polynomials of degree $t = 2$. Hence, since $2 = t < \frac{n-s}{3} = 7/3 = 2.3$ we should be able to correct a single error. Using the method above we want to determine the polynomial $Q(X, Y)$ of the form

$$Q(X,Y) = f_{0,0} + f_{1,0} \cdot X + f_{2,0} \cdot X^2 + f_{3,0} \cdot X^3 + f_{4,0} \cdot X^4 - \left(1 + f_{1,1} \cdot X + f_{2,1} \cdot X^2\right) \cdot Y$$

which passes through the seven given points. Hence we have six variables to determine and we are given seven equations. These equations form the linear system, modulo $q = 101$,

$$\begin{pmatrix} 1 & 1 & 1^2 & 1^3 & 1^4 & -44 \cdot 1 & -44 \cdot 1^2 \\ 1 & 2 & 2^2 & 2^3 & 2^4 & -2 \cdot 2 & -2 \cdot 2^2 \\ 1 & 3 & 3^2 & 3^3 & 3^4 & -25 \cdot 3 & -25 \cdot 3^2 \\ 1 & 4 & 4^2 & 4^3 & 4^4 & -23 \cdot 4 & -23 \cdot 4^2 \\ 1 & 5 & 5^2 & 5^3 & 5^4 & -86 \cdot 5 & -86 \cdot 5^2 \\ 1 & 6 & 6^2 & 6^3 & 6^4 & -83 \cdot 6 & -83 \cdot 6^2 \\ 1 & 7 & 7^2 & 7^3 & 7^4 & -14 \cdot 7 & -14 \cdot 7^2 \end{pmatrix} \cdot \begin{pmatrix} f_{0,0} \\ f_{1,0} \\ f_{2,0} \\ f_{3,0} \\ f_{4,0} \\ f_{1,1} \\ f_{2,1} \end{pmatrix} = \begin{pmatrix} 44 \\ 2 \\ 25 \\ 23 \\ 86 \\ 83 \\ 14 \end{pmatrix}.$$

So we are solving the system

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 57 & 57 \\ 1 & 2 & 4 & 8 & 16 & 97 & 93 \\ 1 & 3 & 9 & 27 & 81 & 26 & 78 \\ 1 & 4 & 16 & 64 & 54 & 9 & 36 \\ 1 & 5 & 25 & 24 & 19 & 75 & 72 \\ 1 & 6 & 36 & 14 & 84 & 7 & 42 \\ 1 & 7 & 49 & 40 & 78 & 3 & 21 \end{pmatrix} \cdot \begin{pmatrix} f_{0,0} \\ f_{1,0} \\ f_{2,0} \\ f_{3,0} \\ f_{4,0} \\ f_{1,1} \\ f_{2,1} \end{pmatrix} = \begin{pmatrix} 44 \\ 2 \\ 25 \\ 23 \\ 86 \\ 83 \\ 14 \end{pmatrix} \pmod{101}.$$

We obtain the solution

$$(f_{0,0}, f_{1,0}, f_{2,0}, f_{3,0}, f_{4,0}, f_{1,1}, f_{2,1}) \leftarrow (20, 84, 49, 11, 0, 67, 0),$$

and hence the two polynomials

$$f_0(X) \leftarrow 20 + 84 \cdot X + 49 \cdot X^2 + 11 \cdot X^3 \text{ and } f_1(X) \leftarrow 1 + 67 \cdot X.$$

We find that

$$f(X) \leftarrow \frac{f_0(X)}{f_1(X)} = 20 + 57 \cdot X + 68 \cdot X^2,$$

which is precisely the polynomial we started with at the beginning of this section. Hence, we have corrected for the error in the transmitted code-word.

19.4. Shamir Secret Sharing

We now return to secret sharing schemes, and in particular the Shamir secret sharing scheme. We suppose we have n parties who wish to share a secret so that no t (or fewer) parties can recover the secret. Hence, this is going to be a $(t+1)$-out-of-n threshold secret sharing scheme.

First we suppose there is a trusted dealer who wishes to share the secret s in $\mathbb{F}_q$. He first generates a secret polynomial $f(X)$ of degree t with $f(0) = s$. That is, he generates random integers f_i in $\mathbb{F}_p$ for $i = 1, \ldots, t$ and sets

$$f(X) = s + f_1 \cdot X + \cdots + f_t \cdot X^t.$$

The trusted dealer then identifies each of the n players by an element in a set $\mathbb{X} \subset \mathbb{F}_q \setminus \{0\}$, for example we could take $\mathbb{X} = \{1, 2, \ldots, n\}$ if the characteristic of $\mathbb{F}_q$ was larger than n. Then if $i \in \mathbb{X}$, party i is given the share $s_i \leftarrow f(i)$. Notice that the vector

$$(s, s_1, \ldots, s_n)$$

is a code-word for a Reed–Solomon code. Also note that if $t+1$ parties come together then they can recover the original polynomial via Lagrange interpolation and hence the secret s. Actually, secret reconstruction can be performed more efficiently by making use of the equation

$$s \leftarrow f(0) = \sum_{i=1}^{n} s_i \cdot \delta_i(0).$$

Hence, for a set $Y \subset \mathbb{X}$, we define the vector r_Y by $r_Y = (r_{x_i,Y})_{x_i \in Y}$ to be the public "recombination" vector, where

$$r_{x_i,Y} = \prod_{x_j \in Y, x_j \neq x_i} \frac{-x_j}{x_i - x_j}.$$

Then, if we obtain a set of shares from a subset $Y \subset \mathbb{X}$ of the players, with $\#Y > t$, we can recover s via the simple summation.

$$s \leftarrow \sum_{x_i \in Y} r_{x_i,Y} \cdot s_i.$$

Also note that if we receive some possible values from a set of parties Y, then we can recover the original secret via the Berlekamp–Welch algorithm for decoding Reed–Solomon codes in the presence of errors, assuming the number of invalid shares is bounded by e where

$$e < t < \frac{\#Y}{3}.$$

Shamir secret sharing is an example of a secret sharing scheme which can be made into a pseudo-random secret sharing scheme, or PRSS. This is a secret sharing scheme which allows the parties to generate a sharing of a random value with almost no interaction. In particular the interaction can be restricted to a set-up phase only.

To define the Shamir pseudo-random secret sharing scheme we take our n parties, which we shall (for sake of concreteness) label by the set $\mathbb{X} = \{1, 2, \ldots, n\}$, and threshold value $t+1$. Then for every subset $A \subset \mathbb{X}$ of size $n - t$ we define the polynomial $f_A(X)$ of degree t by the conditions

$$f_A(0) = 1 \text{ and } f_A(i) = 0 \text{ for all } i \in \mathbb{X} \setminus A.$$

In the initialization phase for our PRSS we create a secret value $r_A \in S$, where S is some key space. For each subset A, the value of r_A is securely distributed to every player in A. The n parties also agree on a public pseudo-random function which is keyed by the secret values r_A,

$$\psi : \begin{cases} S \times S & \longrightarrow & \mathbb{F}_q \\ (r_A, a) & \longmapsto & \psi(r_A, a). \end{cases}$$

Now suppose the parties wish to generate a new secret sharing of a random value. By some means, either by interaction or by prearrangement (e.g. a counter value), they select a public random value $a \in S$. They then generate a random Shamir sharing where the underlying polynomial of degree t is given by

$$f(X) = \sum_{A \subset \mathbb{X}, |A| = n-t} \psi(r_A, a) \cdot f_A(X),$$

where the sum is over all subsets A of size $n - t$. This means that each party i receives the share

$$s_i \leftarrow \sum_{i \in A \subset \mathbb{X}, |A| = n-t} \psi(r_A, a) \cdot f_A(i)$$

where the sum is over all subsets A of size $n-t$ which contain the element i. Finally the random value which is shared, via the Shamir secret sharing scheme, is given by

$$s = \sum_{A \subset \mathbb{X}, |A|=n-t} \psi(r_A, a) \cdot f_A(0) = \sum_{A \subset \mathbb{X}, |A|=n-t} \psi(r_A, a).$$

Other secret sharing schemes can also be turned into PRSSs. Consider the n-out-of-n scheme over $\mathbb{F}_q$ for which the secret is given by $s = s_1 + \cdots + s_n$, where s_i is chosen uniformly at random from the field $\mathbb{F}_q$. This is immediately a PRSS, since to generate a sharing of a random value unknown to any party, each party only needs to generate a random value.

In Chapter 22 we shall require not only a pseudo-random secret sharing, but also a variant called pseudo-random zero sharing, or PRZS, for the Shamir secret sharing scheme. In pseudo-random zero sharing we wish to generate random sharings of the value zero, with respect to a polynomial of degree $2 \cdot t$. The exact reason why the polynomial has to be of degree $2 \cdot t$ will become apparent when we discuss our application in Chapter 22. To enable this extra functionality we require exactly the same set-up as for the PRSS, but now we use a different pseudo-random function,

$$\psi : \begin{cases} S \times S \times \{1, \ldots, t\} & \longrightarrow & \mathbb{F}_q \\ (r_A, x, j) & \longmapsto & \psi(r_A, x, j). \end{cases}$$

Then to create a degree $2 \cdot t$ Shamir secret sharing of zero, the parties pick a number a as before. The underlying polynomial is then given by

$$f(X) = \sum_{A \subset \mathbb{X}, |A|=n-t} \left(\sum_{j=1}^{t} \psi(r_A, a, j) \cdot X^j \cdot f_A(X) \right).$$

Clearly this is a polynomial which shares the zero value, as the polynomial is divisible by X. The share for party i is given by

$$s_i \leftarrow \sum_{A \subset \mathbb{X}, |A|=n-t} \left(\sum_{j=1}^{t} \psi(r_A, a, j) \cdot i^j \cdot f_A(i) \right).$$

19.5. Application: Shared RSA Signature Generation

We shall now present a simple application of a secret sharing scheme, which has applications in the real world. Having introduced digital certificates in Chapter 18, we present an application of secret sharing to distributed RSA signatures. Suppose a company is setting up a certificate authority to issue RSA signed certificates to its employees to enable them to access various corporate services. It considers the associated RSA private key to be highly sensitive, after all if the private key was compromised then the company's entire corporate infrastructure could also be compromised. Suppose the public key is (N, e) and the private key is d.

The company decides that to mitigate the risk it will divide the private key into three shares and place the three shares on three different continents. Thus, for example, there will be one server in Asia, one in America and one in Europe. As soon as the RSA key is generated, the company generates three integers d_1, d_2 and d_3 such that

$$d = d_1 + d_2 + d_3 \pmod{\phi(N)}.$$

The company then removes all knowledge of d and places d_1 on a secure computer in Asia, d_2 on a secure compute in America and d_3 on a secure computer in Europe.

Now an employee wishes to obtain a digital certificate. This is essentially the RSA signature on a (probably hashed) string m. The employee simply sends the string m to the three computers,

which respond with

$$s_i \leftarrow m^{d_i} \text{ for } i = 1, 2, 3.$$

The valid RSA signature is then obtained by multiplying the three shares together, i.e.

$$s \leftarrow s_1 \cdot s_2 \cdot s_3 = m^{d_1+d_2+d_3} = m^d.$$

Here, we have used the fact that the RSA function is multiplicatively homomorphic.

This scheme appears to solve the problem of not putting the master signature key in only one location. However, the employee now needs the three servers to be online in order to obtain his certificate. It would be much nicer if only two had to be online, since then the company could cope with outages of servers. The problem is that the above scheme essentially implements a 3-out-of-3 secret sharing scheme, whereas what we want is a 2-out-of-3. Clearly, we need to apply something along the lines of Shamir secret sharing. However, the problem is that the number $\phi(N)$ needs to be kept secret, and the denominators in the Lagrange interpolation formulae may not be coprime to $\phi(N)$.

There have been many solutions proposed to the above problem of threshold RSA, however, the most elegant and simple is due to Shoup. Suppose we want a t-out-of-n sharing of the RSA secret key d, where we assume that e is chosen so that it is a prime and $e > n$. We adapt the Shamir scheme as follows: Firstly a polynomial of degree $t-1$ is chosen, by selecting f_i modulo $\phi(N)$ at random, to obtain

$$f(X) = d + f_1 \cdot X + \cdots + f_{t-1} \cdot X^{t-1}.$$

Then each server is given the share $d_i = f(i)$. The number of parties n is assumed to be fixed and we define Δ to be the constant $\Delta = n!$.

Now suppose a user wishes to obtain a signature on the message m, i.e it wants to compute $m^d \pmod N$. It sends m to each server, which then computes the signature fragment as

$$s_i = m^{2 \cdot \Delta \cdot d_i} \pmod N.$$

These signature fragments are then sent back to the user. Suppose now that the user obtains fragments back from a subset $Y = \{i_1, \ldots, i_t\} \subset \{1, \ldots, n\}$, of size greater than or equal to t. This set defines a "recombination" vector $\mathbf{r}_Y = (r_{i_j,Y})_{i_j \in Y}$ defined by

$$r_{i_j,Y} \leftarrow \prod_{i_k \in Y, i_j \neq i_k} \frac{-i_k}{i_j - i_k}.$$

We really want to be able to compute this modulo $\phi(N)$, but that is impossible since $\phi(N)$ is not known to any of the participants. In addition the denominator may not be invertible modulo $\phi(N)$. However, we note that the denominator in the above divides Δ and so we have that $\Delta \cdot r_{i_j,Y} \in \mathbb{Z}$. Hence, the user can compute

$$\sigma \leftarrow \prod_{i_j \in Y} {s_{i_j}}^{2 \cdot \Delta \cdot r_{i_j,Y}} \pmod N.$$

We find that this is equal to

$$\sigma = \left(m^{4 \cdot \Delta^2}\right)^{\sum_{i_j \in Y} r_{i_j,Y} \cdot d_{i_j}} = m^{4 \cdot \Delta^2 \cdot d} \pmod N,$$

with the last equality working due to Lagrange interpolation modulo $\phi(N)$. From this partial signature we need to recover the real signature. To do this we use the fact that we have assumed that $e > n$ and that e is a prime. These latter two facts mean that e is coprime to $4 \cdot \Delta^2$, and so via the extended Euclidean algorithm we can compute integers u and v such that

$$u \cdot e + v \cdot 4 \cdot \Delta^2 = 1,$$

from which the signature is computed as

$$s \leftarrow m^u \cdot \sigma^v \pmod{N}.$$

That s is the valid RSA signature for this public/private key pair can be verified since

$$\begin{aligned} s^e &= (m^u \cdot \sigma^v)^e = m^{e \cdot u} \cdot m^{4 \cdot e \cdot v \cdot \Delta^2 \cdot d}, \\ &= m^{u \cdot e + 4 \cdot v \cdot \Delta^2} = m. \end{aligned}$$

One problem with the protocol as we have described it is that the signature shares s_i may be invalid. See the paper by Shoup in the Further Reading section to see how this problem can be removed using zero-knowledge proofs.

Chapter Summary

- We have defined the general concept of secret sharing schemes and shown how these can be constructed, albeit inefficiently, for any access structure.
- We have introduced Reed–Solomon error-correcting codes and presented the Berlekamp–Welch decoding algorithm.
- We presented Shamir's secret sharing scheme, which produces a highly efficient, and secure, secret sharing scheme in the case of threshold access structures.
- We extended the Shamir scheme to give both pseudo-random secret sharing and pseudo-random zero sharing.
- Finally we showed how one can adapt the Shamir scheme to enable the creation of a threshold RSA signature scheme.

Further Reading

Shamir's secret sharing scheme is presented in his short ACM paper from 1979. Shoup's threshold RSA scheme is presented in his Eurocrypt 2000 paper; this paper also explains the occurrence of the Δ^2 term in the above discussion, rather than a single Δ term. A good description of secret sharing schemes for general access structures, including some relatively efficient constructions, is presented in the relevant chapter in Stinson's book.

A. Shamir. *How to share a secret.* Communications of the ACM, **22**, 612–613, 1979.

V. Shoup. *Practical threshold signatures.* In Advances in Cryptology – Eurocrypt 2000, LNCS 1807, 207–220, Springer, 2000.

D. Stinson. *Cryptography: Theory and Practice.* Third Edition. CRC Press, 2005.

CHAPTER 20

Commitments and Oblivious Transfer

Chapter Goals

- To present two protocols which are carried out between mutually untrusting parties.
- To introduce commitment schemes and give simple examples of efficient implementations.
- To introduce oblivious transfer, and again give simple examples of how this can be performed in practice.

20.1. Introduction

In this chapter we shall examine a number of more advanced cryptographic protocols which enable higher-level services to be created. We shall particularly focus on protocols for

- commitment schemes,
- oblivious transfer.

Whilst there is a large body of literature on these protocols, we shall keep our feet on the ground and focus on protocols which can be used in real life to achieve practical higher-level services. It turns out that these two primitives are in some sense the most basic atomic cryptographic primitives which one can construct.

Up until now we have looked at cryptographic schemes and protocols in which the protocol participants are honest, and we are trying to protect their interests against an external adversary. However, in the real world we often need to interact with people who we do not necessarily trust. In this chapter we examine two types of protocol which are executed between two parties, each of whom may want to cheat in some way. The simplistic protocols in this chapter will form the building blocks on which more complicated protocols will be built in Chapters 21 and 22. We start by focusing on commitment schemes, and then we pass to oblivious transfer.

20.2. Commitment Schemes

Suppose Alice wishes to play "paper-scissors-stone" over the telephone with Bob. The idea of this game is that Alice and Bob simultaneously choose one of the set {`paper`,`scissors`,`stone`}. Then the outcome of the game is determined by the rules:

- `Paper` wraps `stone`. Hence if Alice chooses `paper` and Bob chooses `stone` then Alice wins.
- `Stone` blunts `scissors`. Hence if Alice chooses `stone` and Bob chooses `scissors` then Alice wins.
- `Scissors` cut `paper`. Hence if Alice chooses `scissors` and Bob chooses `paper` then Alice wins.

If both Alice and Bob choose the same item then the game is declared a draw. When conducted over the telephone we have the problem that whoever goes first is going to lose the game.

One way around this is for the party who goes first to "commit" to their choice, in such a way that the other party cannot determine what was committed to. Then the two parties can reveal their choices, with the idea that the other party can then verify that the revealing party has

not altered its choice between the commitment and the revealing stage. Such a system is called a *commitment scheme*. An easy way to do this is to use a cryptographic hash function as follows:

$$\begin{aligned} A &\longrightarrow B : h_A = H(R_A \| \texttt{paper}), \\ B &\longrightarrow A : \texttt{scissors}, \\ A &\longrightarrow B : R_A, \texttt{paper}. \end{aligned}$$

At the end of the protocol Bob needs to verify that the h_A sent by Alice is equal to $H(R_A\|\texttt{paper})$. If the values agree he knows that Alice has not cheated. The result of this protocol is that Alice loses the game since `scissors` cut `paper`.

Let us look at the above from Alice's perspective. She first commits to the value `paper` by sending Bob the hash value h_A. This means that Bob will not be able to determine that Alice has committed to the value `paper`, since Bob does not know the random value of R_A used and Bob is unable to invert the hash function. The fact that Bob cannot determine what value was committed to is called the concealing or hiding property of a commitment scheme.

As soon as Bob sends the value `scissors` to Alice, she knows she has lost but is unable to cheat, since to cheat she would need to come up with a different value of R_A, say R'_A, which satisfied

$$H(R_A\|\texttt{paper}) = H(R'_A\|\texttt{stone}).$$

But this would mean that Alice could find collisions in the hash function, which for a suitably chosen hash function is believed to be impossible. Actually we require that the hash function is second preimage resistant in this case. This property of the commitment scheme, that Alice cannot change her mind after the commitment procedure, is called binding.

Let us now study these properties of concealing and binding in more detail. Recall that an encryption function has information-theoretic security if an adversary with infinite computing power could not break the scheme, whilst an encryption function is called computationally secure if it is only secure when faced with an adversary with polynomially bounded computing power. A similar division can be made with commitment schemes, but now we have two security properties, namely concealing and binding. One property protects the interests of the sender, and one property protects the interests of the receiver. To simplify our exposition we shall denote our abstract commitment scheme by a public algorithm, $c = C(x, r)$ which takes a value $x \in \mathbb{P}$ and some randomness $r \in \mathbb{R}$ and produces a commitment $c \in \mathbb{C}$. To decomitment the commiter simply reveals the values of x and r. The receiver then checks that the two values produce the original commitment.

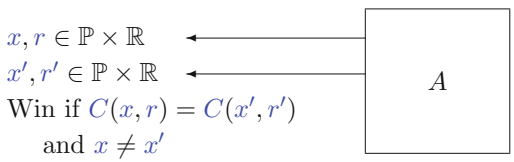


FIGURE 20.1. Commitment scheme: binding game

Definition 20.1 (Binding)**.** *A commitment scheme is said to be information-theoretically (resp. computationally)* binding *if no infinitely powerful (resp. computationally bounded) adversary can win the following game.*

- *The adversary outputs values $x \in \mathbb{P}$ and $r \in \mathbb{R}$.*
- *The adversary must then output a value $x' \neq x$ and a value $r' \in \mathbb{R}$ such that*

$$C(x, r) = C(x', r').$$

This game is given graphically in Figure 20.1. If the commitment scheme Π is *computationally* binding then we can define an advantage statement which is defined as

$$\mathrm{Adv}_{\Pi}^{\mathsf{bind}} = \Pr[A \text{ wins the binding game}].$$

For information-theoretically binding schemes such an advantage will be zero by definition.

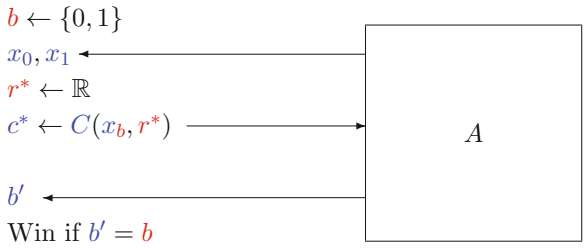


FIGURE 20.2. Commitment scheme: concealing game

Definition 20.2 (Concealing)**.** *A commitment scheme is said to be information-theoretically (resp. computationally)* concealing *if no infinitely powerful (resp. computationally bounded) adversary can win the following game.*

- *The adversary outputs two messages x_0 and x_1 of equal length.*
- *The challenger generates $r^* \in \mathbb{R}$ at random and a random bit $b \in \{0,1\}$.*
- *The challenger computes $c^* = C(x_b, r^*)$ and passes c^* to the adversary.*
- *The adversary's goal now is to guess the bit b.*

This game is defined in Figure 20.2. Just as with the binding property, if the commitment scheme Π is *computationally* concealing then we can define an advantage statement which is defined as

$$\mathrm{Adv}_{\Pi}^{\mathsf{conceal}} = 2 \cdot \left| \Pr[A \text{ wins the concealing game}] - \frac{1}{2} \right|.$$

For information-theoretically concealing schemes such an advantage will be zero by definition. Notice that this definition of concealing is virtually identical to our definition of indistinguishability of encryptions. A number of results trivially follow from these two definitions.

Lemma 20.3. *There exists no scheme which is both information-theoretically concealing and binding.*

PROOF. Suppose we have a scheme which is both information-theoretically concealing and binding, and suppose the committer makes a commitment $c \leftarrow C(x, r)$. Since it is information-theoretically concealing there must exist values x' and r' such that $c = C(x', r')$; otherwise an infinitely powerful receiver could break the concealing property. But if Alice is also infinitely powerful this means she can break the binding property as well. □

Lemma 20.4. *Using the commitment scheme defined as*

$$c \leftarrow H(r \| m),$$

for a random value r, the committed value m and some cryptographic hash function H is at best

- *computationally binding,*
- *information-theoretically concealing.*

PROOF. All cryptographic hash functions we have met are only computationally secure against preimage resistance and second preimage resistance. The binding property of the above scheme

is only guaranteed by the second preimage resistance of the underlying hash function. Hence, the binding property is only computationally secure.

The concealing property of the above scheme is only guaranteed by the preimage resistance of the underlying hash function. Hence, the concealing property looks like it should be only computationally secure. However, if we assume that the value r is chosen from a suitably large set, then the fact that the hash function should have many collisions works in our favour and in practice we should obtain something "close" to information-theoretic concealing. On the other hand if we assume that H is a random oracle, then the commitment scheme is clearly information-theoretically concealing. □

We now turn to three practical commitment schemes which occur in various real-world protocols. All are based on a finite abelian group G of prime order q, which is generated by g. Two of the schemes will also require another generator $h \in \langle g \rangle$, where the discrete logarithm of h to the base g is unknown by any user in the system. To generate g and h we need to ensure that no one knows the discrete logarithm, and hence it needs to be done in a verifiably random manner.

Verifiably random generation of g and h is quite easy to ensure, for example for a finite field $\mathbb{F}_p^*$ with q dividing $p-1$ we create g as follows (with a similar procedure being used to determine h):

- $r \leftarrow \mathbb{Z}$.
- $f \leftarrow H(r) \in \mathbb{F}_p^*$ for some cryptographic hash function H.
- $g \leftarrow f^{(p-1)/q} \pmod{p}$.
- If $g = 1$ then return to the first stage, else output (r, g).

This generates a random element of the subgroup of $\mathbb{F}_p^*$ of order q, with the property that it is generated verifiably at random since one outputs the seed r used to generate the random element. Thus anyone who wishes to verify that g was generated in the above manner can use r to rerun the algorithm. Since we have used a cryptographic hash function H we do not believe that it is feasible for anyone to construct an r which produces a group element whose discrete logarithm is known with respect to some other group element.

Given g, h we define two commitment schemes, $B(x)$ and $B_a(x)$, to commit to an integer x modulo q, and one, $E_a(x)$, to commit to an element $x \in \langle g \rangle$.

$$\begin{aligned} B(x) &= g^x, \\ E_a(x) &= (g^a, x \cdot h^a), \\ B_a(x) &= h^x \cdot g^a, \end{aligned}$$

where a is a random integer modulo q. To reveal the commitments the user publishes the value x in the first scheme and the pair (a, x) in the second and third schemes. The value a is called the blinding value, since it blinds the value of the commitment x even to a computationally unbounded adversary. The scheme given by $B_a(x)$ is called Pedersen's commitment scheme.

Lemma 20.5. *The commitment scheme $B(x)$ is information-theoretically binding.*

Proof. Suppose Alice having published $c = B(x) = g^x$ wishes to change her mind as to which element of $\mathbb{Z}/q\mathbb{Z}$ she wants to commit to. Alas, for Alice no matter how much computing power she has there is mathematically only one element in $\mathbb{Z}/q\mathbb{Z}$, namely x, which is the discrete logarithm of the commitment c to the base g. Hence, the scheme is clearly information-theoretically binding. □

Note that this commitment scheme does not meet our strong definition of security for the concealing property, in any way; after all it is deterministic. If the space of values from which x is selected

is large, then this commitment scheme could meet a weaker security definition related to a one-way-like property. We leave the reader to produce a suitable definition for a one-way concealing property, and show that $B(x)$ meets this definition.

Lemma 20.6. *The commitment scheme $E_a(x)$ is information-theoretically binding and computationally concealing.*

PROOF. This scheme is exactly ElGamal encryption with respect to a public key h, for which *no one* knows the associated private key. Indeed any IND-CPA secure public key encryption scheme can be used in this way as a commitment scheme.

The underlying IND-CPA security implies that the resulting commitment scheme is computationally concealing, whilst the fact that the decryption is unique implies that the commitment scheme is information-theoretically binding. □

Lemma 20.7. *The Pedersen commitment scheme, given by $B_a(x)$, is computationally binding and information-theoretically concealing.*

PROOF. Suppose the adversary, after having committed to $c \leftarrow B_a(x) = h^x \cdot g^a$ wishes to change her mind, so as to commit to y instead. So the adversary outputs another pair (y, b) such that $c = h^y \cdot g^b$. However, given these two values we can extract the discrete logarithm of h with respect to g via

$$\frac{a-b}{y-x}.$$

Thus any algorithm which breaks the binding property can be turned into an algorithm which solves discrete logarithms in the group G.

We now turn to the concealing property. It is clear that this is information-theoretically concealing since an all-powerful adversary could extract the discrete logarithm of h with respect to g and then *any* committed value c can be opened to *any* message x. □

We end this section by noticing that the two discrete-logarithm-based commitment schemes we have given possess the homomorphic property:

$$\begin{aligned} B(x_1) \cdot B(x_2) &= g^{x_1} \cdot g^{x_2} \\ &= g^{x_1+x_2} \\ &= B(x_1 + x_2), \\ B_{a_1}(x_1) \cdot B_{a_2}(x_2) &= h^{x_1} \cdot g^{a_1} \cdot h^{x_2} \cdot g^{a_2} \\ &= h^{x_1+x_2} \cdot g^{a_1+a_2} \\ &= B_{a_1+a_2}(x_1 + x_2). \end{aligned}$$

We shall use this additively homomorphic property when we discuss an electronic voting protocol at the end of Chapter 21.

20.3. Oblivious Transfer

We now consider another type of basic protocol, called oblivious transfer or OT for short. This is another protocol which is run between two distrusting parties, a sender and a receiver. In its most basic form the sender has two secret messages as input, m_0 and m_1; the receiver has as input a single bit b. The goal of an OT protocol is that at the end of the protocol the sender should not learn the value of the receiver's input b. However, the receiver should learn the value of m_b but should learn nothing about m_{1-b}. Such a protocol is often called a 1-out-of-2 OT, since the receiver learns one of the two inputs of the sender. Such a protocol is depicted in Figure 20.3. One

can easily generalize this concept to a k-out-of-n OT, but as it is we will only be interested in the simpler case.

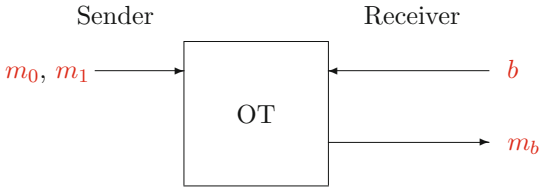


FIGURE 20.3. A 1-out-of-2 OT

We present a scheme which allows us to perform a 1-out-of-2 oblivious transfer of two arbitrary bit strings m_0, m_1 of equal length. The scheme is based on an IND-CPA version of DHIES, where we use simply the exclusive-or of the plaintext with a hash of the underlying Diffie–Hellman key to encryt the payload.

We take a standard discrete-logarithm-based public/private key pair $(h \leftarrow g^x, x)$, where g is a generator of cyclic finite abelian group G of prime order q. We will require a hash function H from G to bit strings of length n. Then to encrypt messages m of length n we compute, for a random $k \in (\mathbb{Z}/q\mathbb{Z})$,

$$c = (c_1, c_2) \leftarrow \left(g^k, m \oplus H(h^k)\right).$$

To decrypt we compute

$$c_2 \oplus H({c_1}^x) = m \oplus H(g^{kx}) = m \oplus H(h^k) = m.$$

It can easily be shown that the above scheme is semantically secure under chosen plaintext attacks (i.e. passive attacks) in the random oracle model.

The idea behind our oblivious transfer protocol is for the receiver to create two public keys h_0 and h_1, for only one of which does he know the corresponding secret key. If the receiver knows the secret key for h_b, where b is the bit he is choosing, then he can decrypt for messages encrypted under this key, but not decrypt under the other key. The sender then only needs to encrypt his messages with the two keys. Since the receiver only knows one secret key he can only decrypt one of the messages.

To implement this idea concretely, the sender first selects a random element c in G; it is important that the receiver does not know the discrete logarithm of c with respect to g. This value is then sent to the receiver. The receiver then generates two public keys, according to his bit b, by first generating $x \in (\mathbb{Z}/q\mathbb{Z})$ and then computing

$$h_b \leftarrow g^x, \ h_{1-b} \leftarrow c/h_b.$$

Notice that the receiver knows the underlying secret key for h_b, but he does not know the secret key for h_{1-b} since he does not know the discrete logarithm of c with respect to g. These two public key values are then sent to the sender. The sender then encrypts message m_0 using the key h_0 and message m_1 using key h_1, i.e. the sender computes

$$c_0 \leftarrow \left(g^{k_0}, m_0 \oplus H({h_0}^{k_0})\right),$$
$$c_1 \leftarrow \left(g^{k_1}, m_1 \oplus H({h_1}^{k_1})\right),$$

for two random integers k_0, $k_1 \in (\mathbb{Z}/q\mathbb{Z})$. These two ciphertexts are then sent to the receiver who then decrypts the bth one using his secret key x.

From the above description we can obtain some simple optimizations. Firstly, the receiver does not need to send both h_0 and h_1 to the sender, since the sender can always compute h_1 from h_0 by

computing c/h_0. Secondly, we can use the same value of $k = k_0 = k_1$ in the two encryptions. We thus obtain the following oblivious transfer protocol:

Sender		Receiver
$c \leftarrow G$	$\xrightarrow{c}$	
		$x \leftarrow (\mathbb{Z}/q\mathbb{Z})$
		$h_b \leftarrow g^x$
	$\xleftarrow{h_0}$	$h_{1-b} \leftarrow c/h_b$
$h_1 \leftarrow c/h_0$		
$k \leftarrow (\mathbb{Z}/q\mathbb{Z})$		
$c_1 \leftarrow g^k$		
$e_0 \leftarrow m_0 \oplus H({h_0}^k)$		
$e_1 \leftarrow m_1 \oplus H({h_1}^k)$	$\xrightarrow{c_1,e_0,e_1}$	
		$m_b \leftarrow e_b \oplus H({c_1}^x)$.

So does this respect the two conflicting security requirements of the participants? First, note that the sender cannot determine the hidden bit b of the receiver since the value h_0 sent from the receiver is simply a random element in G. Then we note that the receiver can learn nothing about m_{1-b} since to do this they would have to be able to compute the output of H on the value h_{1-b}^k, which would imply contradicting the fact that H acts as a random oracle or being able to solve the Diffie–Hellman problem in the group G.

Chapter Summary

- We introduced the idea of protocols between mutually untrusting parties, and introduced commitment and oblivious transfer as two simple examples of such protocols.
- A commitment scheme allows one party to bind themselves to a value, and then reveal it later.
- A commitment scheme needs to be both binding and concealing. Efficient schemes exist which are either information-theoretically binding or information-theoretically concealing, but not both.
- An oblivious transfer protocol allows a sender to send one of two messages to a recipient, but she does not know which message is actually received. The receiver also learns nothing about the other message which was sent.

Further Reading

The above oblivious transfer protocol originally appeared in a slightly modified form in the paper by Bellare and Micali. The paper by Naor and Pinkas discusses a number of optimizations of the oblivious transfer protocol which we presented above. In particular it presents mechanisms to efficiently perform 1-out-of-N oblivious transfer. The papers by Blum and Shamir et al. provide some nice early ideas related to commitment schemes.

M. Bellare and S. Micali. *Non-interactive oblivious transfer and applications.* In Advances in Cryptology – Crypto 1989, LNCS 435, 547–557, Springer, 1990.

M. Blum. *Coin flipping by telephone.* SIGACT News, **15**, 23–27, 1983.

M. Naor and B. Pinkas. *Efficient oblivious transfer protocols.* In SIAM Symposium on Discrete Algorithms – SODA 2001, 448–457, SIAM, 2001.

A. Shamir, R. Rivest and L. Adleman. *Mental Poker.* The Mathematical Gardner, 37–43, Prindle, Weber and Schmidt, 1981.

CHAPTER 21

Zero-Knowledge Proofs

Chapter Goals

- To introduce zero-knowledge proofs.
- To explain the notion of simulation.
- To introduce Sigma protocols.
- To explain how these can be used in a voting protocol.

21.1. Showing a Graph Isomorphism in Zero-Knowledge

Suppose Alice has a password and wants to log in to a website run by Bob, but she does not quite trust the computer Bob is using to verify the password. If she just sends the password to Bob then Bob's computer will learn the whole password. To get around this problem one often sees websites that ask for the first, fourth and tenth letter of a password one time, and then maybe the first, second and fifth the second time and so on. In this way Bob's computer only learns three letters at a time. So the password can be checked but in each iteration of checking only three letters are leaked. It clearly would be better if Bob could verify that Alice has the password in such a way that Alice never has to reveal *any* of the password to Bob. This is the problem this chapter will try to solve.

So we suppose that Alice wants to convince Bob that she knows something without Bob finding out exactly what Alice knows. This apparently contradictory state of affairs is dealt with using zero-knowledge proofs. In the literature of zero-knowledge proofs, the role of Alice is called the prover, since she wishes to prove something, whilst the role of Bob is called the verifier, since he wishes to verify that the prover actually knows something. Often, and we shall also follow this convention, the prover is called Peggy and the verifier is called Victor.

The classic example of a zero-knowledge proof is based on the graph isomorphism problem. Given two graphs G_1 and G_2, with the same number of vertices, we say that the two graphs are isomorphic if there is a relabelling (i.e. a permutation) of the vertices of one graph which produces the second graph. This relabelling ϕ is called a graph isomorphism, which is denoted by

$$\phi : G_1 \longrightarrow G_2.$$

It is a computationally hard problem to determine a graph isomorphism between two graphs. As a running example consider the two graphs in Figure 21.1, linked by the permutation $\phi = (1, 2, 4, 3)$.

Suppose Peggy knows the graph isomorphism ϕ between two public graphs G_1 and G_2, so we have $G_2 = \phi(G_1)$. We call ϕ the prover's private input, whilst the graphs G_1 and G_2 are the public or common input. Peggy wishes to convince Victor that she knows the graph isomorphism, without revealing to Victor the precise nature of the graph isomorphism. This is done using the following zero-knowledge proof.

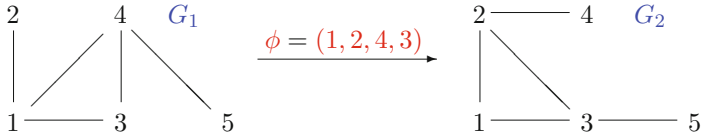


FIGURE 21.1. Example graph isomorphism

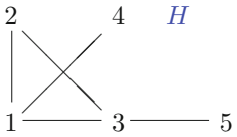


FIGURE 21.2. Peggy's committed graph

- Peggy takes the graph G_2 and applies a secret random permutation ψ to the vertices of G_2 to produce another isomorphic graph $H \leftarrow \psi(G_2)$. In our running example we take $\psi = (1, 2)$; the isomorphic graph H is then given by Figure 21.2.
- Peggy now publishes H as a **commitment**. She of course knows the following secret graph isomorphisms

$$\begin{aligned} \phi &: G_1 \longrightarrow G_2, \\ \psi &: G_2 \longrightarrow H, \\ \psi \circ \phi &: G_1 \longrightarrow H. \end{aligned}$$

- Victor now gives Peggy a **challenge**. He selects[1] $b \in \{1, 2\}$ and asks for the graph isomorphism between G_b and H
- Peggy now gives her **response** by returning either $\chi = \psi$ or $\chi = \psi \circ \phi$, depending on the value of b.
- Victor now verifies whether $\chi(G_b) = H$.

The transcript of the protocol then looks like

$$\begin{aligned} P &\longrightarrow V : H, \\ V &\longrightarrow P : b, \\ P &\longrightarrow V : \chi. \end{aligned}$$

In our example if Victor chooses $b = 2$ then Peggy simply needs to publish ψ. However, if Victor chooses $b = 1$ then Peggy publishes

$$\psi \circ \phi = (1, 2) \circ (1, 2, 4, 3) = (2, 4, 3).$$

We can then see that $(2, 4, 3)$ is the permutation which maps graph G_1 onto graph H. But to compute this we needed to know the hidden isomorphism ϕ. Thus when $b = 2$ Victor is checking whether Peggy is honest in her commitment, whilst if $b = 1$ he is checking whether Peggy is honest in her claim to know the isomorphism from G_1 to G_2.

If Peggy does not know the graph isomorphism ϕ then, before Victor gives his challenge, she will need to know the graph G_b which Victor is going to pick. Hence, if Peggy is cheating she will

[1] Since it is seleted by Victor we denote the value b in blue.

only be able to respond to correctly to Victor fifty percent of the time. So, repeating the above protocol a number of times, a non-cheating Peggy will be able to convince Victor that she really does know the graph isomorphism, with a small probability that a cheating Peggy will be able to convince Victor incorrectly.

Now we need to determine whether Victor learns anything from running the protocol, i.e. is Peggy's proof really zero-knowledge? We first notice that Peggy needs to produce a different value of H on every run of the protocol, otherwise Victor can trivially cheat. We assume therefore that this does not happen.

One way to see whether Victor has learnt something after running the protocol is to look at the transcript of the protocol and ask after having seen the transcript whether Victor has gained any knowledge, or for that matter whether anyone looking at the protocol but not interacting learns anything. One way to see that Victor has not learnt anything is to see that Victor could have written down a valid protocol transcript without interacting with Peggy at all. Hence, Victor cannot use the protocol transcript to convince someone else that he knows Peggy's secret isomorphism. He cannot even use the protocol transcript to convince another party that Peggy knows the secret graph isomorphism ϕ.

Victor can produce a valid protocol transcript using the following *simulation*:

- $b \leftarrow \{1, 2\}$.
- Generate a random isomorphism χ of the graph G_b to produce the graph H.
- Output the transcript

$$P \longrightarrow V : H,$$
$$V \longrightarrow P : b,$$
$$P \longrightarrow V : \chi.$$

Hence, the interactive nature of the protocol is what provides the "proof" in the zero-knowledge proof. We remark that the three-pass system of

$$\text{commitment} \longrightarrow \text{challenge} \longrightarrow \text{response}$$

is the usual characteristic of such protocols when deployed in practice. Notice how this is similar to the signature schemes we discussed in Section 16.5.4. This is not coincidental, as we shall point out below.

Clearly two basic properties of an interactive proof system are

- **Completeness:** If Peggy really knows the thing being proved and follows the protocol, then Victor should accept her proof with probability one.
- **Soundness:** If Peggy does not know the thing being proved, then whatever she does, Victor should only have a small probability of actually accepting the proof.

Just as with commitment schemes we can divide zero-knowledge protocols into categories depending on whether they are secure with respect to computationally bounded or unbounded adversaries. We usually assume that Victor is a polynomially bounded party, whilst Peggy is unbounded[2]. In the above protocol based on graph isomorphism we saw that the soundness probability was equal to one half. Hence, we needed to repeat the protocol a number of times to improve this to something close to one.

The zero-knowledge property we have already noted is related to the concept of a simulation. Suppose the set of valid transcripts (produced by true protocol runs) is denoted by $\mathcal{V}$, and let the set of possible simulations be denoted by $\mathcal{S}$. The security is therefore related to how much the set $\mathcal{V}$ is like the set $\mathcal{S}$. A zero-knowledge proof is said to have perfect zero-knowledge if the two

[2]Although in many of our examples the existence of a witness is certain and hence we might as well assume that Peggy knows the witness already and is bounded.

sets $\mathcal{V}$ and $\mathcal{S}$ are essentially identical, in which case we write $\mathcal{V} = \mathcal{S}$. If the two sets have "small" statistical distance[3], but cannot otherwise be distinguished by an all-powerful adversary, we say we have statistical zero-knowledge, and we write $\mathcal{V} \approx_s \mathcal{S}$. If the two sets are only indistinguishable by a computationally bounded adversary we say that the zero-knowledge proof has computational zero-knowledge, and we write $\mathcal{V} \approx_c \mathcal{S}$.

21.2. Zero-Knowledge and $\mathcal{NP}$

So the question arises as to what can be shown in zero-knowledge. Above we showed that the knowledge of whether two graphs are isomorphic can be shown in zero-knowledge. Thus the decision problem of `Graph Isomorphism` lies in the set of all decision problems which can be proven in zero-knowledge. But `Graph Isomorphism` is believed to lie between the complexity classes $\mathcal{P}$ and $\mathcal{NP}$-complete, i.e. it can neither be solved in polynomial time, nor is it $\mathcal{NP}$-complete.

We can think of $\mathcal{NP}$ problems as those problems for which there is a witness (or proof) which can be produced by an all-powerful prover, but for which a polynomially bounded verifier is able to verify the proof. However, for the class of $\mathcal{NP}$ problems the prover and the verifier do not interact, i.e. the proof is produced and then the verifier verifies it.

If we allow interation then something quite amazing happens. Consider an all powerful prover who interacts with a polynomially bounded verifier. We wish the prover to convince the verifier of the validity of some statement. This is exactly what we had in the previous section except that we only require the completeness and soundness properties, i.e. we do not require the zero-knowledge property. The decision problems which can be proven to be true in such a manner form the complexity class of *interactive proofs*, or $\mathcal{IP}$. It can be shown that the complexity class $\mathcal{IP}$ is equal to the complexity class $\mathcal{PSPACE}$, i.e. the set of all decision problems which can be solved using polynomial space. It is widely believed that $\mathcal{NP} \subsetneq \mathcal{PSPACE}$, which implies that having interaction really gives us something extra.

So what happens to interactive proofs when we add the zero-knowledge requirement? We can define a complexity class $\mathcal{CZK}$ of all decision problems whose solutions can be verified using a computational zero-knowledge proof. We have already shown that the problem of `Graph Isomorphism` lies in $\mathcal{CZK}$, but this might not include all of the $\mathcal{NP}$ problems. However, since 3-colourability is $\mathcal{NP}$-complete, we have the following elegant proof that $\mathcal{NP} \subset \mathcal{CZK}$,

Theorem 21.1. *The problem of* 3*-colourability of a graph lies in* $\mathcal{CZK}$*, assuming a computationally concealing commitment scheme exists.*

Proof. Consider a graph $G = (V, E)$ in which the prover knows a colouring ψ of G, i.e. a map $\psi : V \to \{1, 2, 3\}$ such that $\psi(v_1) \neq \psi(v_2)$ if $(v_1, v_2) \in E$. The prover first selects a commitment scheme $C(x, r)$ and a random permutation π of the set $\{1, 2, 3\}$. Note that the function $\pi(\psi(v))$ defines another three-colouring of the graph. Now the prover commits to this second three-colouring by sending to the verifier the commitments

$$c_i = C\left(\pi(\psi(v_i)), r_i\right) \text{ for all } v_i \in V.$$

The verifier then selects a random edge $(v_i, v_j) \in E$ and sends this to the prover. The prover now decommits to the commitment, by returning the values of

$$\pi(\psi(v_i)) \text{ and } \pi(\psi(v_j)),$$

and the verifier checks that

$$\pi(\psi(v_i)) \neq \pi(\psi(v_j)).$$

We now turn to the three required properties of a zero-knowledge proof.

[3]See Chapter 7.

Completeness: The above protocol is complete since any valid prover will get the verifier to accept with probability one.
Soundness: If we have a cheating prover, then at least one edge is invalid, and with probability at least $1/|E|$ the verifier will select an invalid edge. Thus with probability at most $1 - 1/|E|$ a cheating prover will get a verifier to accept. By repeating the above proof many times one can reduce this probability to as low a value as we require.
Zero-Knowledge: Assuming the commitment scheme is computationally concealing, the obvious simulation and the real protocol will be computationally indistinguishable. □

Notice that this is a very powerful result. It says that virtually any statement which is likely to come up in cryptography can be proved in zero-knowledge. Clearly the above proof would not provide a practical implementation, but at least we know that very powerful tools can be applied. In the next section we turn to proofs that can be applied in practice. But before doing that we note that the above result can be extended even further.

Theorem 21.2. *If one-way functions exist then $\mathcal{CZK} = \mathcal{IP}$, and hence $\mathcal{CZK} = \mathcal{PSPACE}$.*

21.3. Sigma Protocols

One can use a zero-knowledge proof of possession of some secret as an identification scheme. The secret in the identification scheme will be the hidden information, or witness, e.g. the graph isomorphism in our previous example. Then we use the zero-knowledge protocol to prove that the person knows the isomorphism, without revealing anything about it. The trouble with the above protocol for graph isomorphisms is that it is not very practical. The data structures required are very large, and the protocol needs to be repeated a large number of times before Victor is convinced that Peggy really knows the secret.

This is exactly what a Sigma protocol provides. It is a three-move protocol: the prover goes first (in the commitment phase), then the verifier responds (with the challenge), and finally the prover provides the final response; the verifier is then able to verify the proof. This is exactly like our graph isomorphism proof earlier. But for Sigma protocols we make some simplifying assumptions; in particular we assume that the verifier is honest (in that he will always follow the protocol correctly).

21.3.1. Schnorr's Identification Protocol: In essence we have already seen a Sigma protocol which has better bandwidth and error properties when we discussed Schnorr signatures in Chapter 16. Suppose Peggy's secret is now the discrete logarithm x of y with respect to g in some finite abelian group G of prime order q. To create an identification protocol, we want to show in zero-knowledge that Peggy knows the value of x. The protocol for proof of knowledge now goes as follows

$$\begin{aligned} &P \longrightarrow V : r \leftarrow g^k \text{ for a random } k \leftarrow \mathbb{Z}/q\mathbb{Z}, \\ &V \longrightarrow P : e \leftarrow \mathbb{Z}/q\mathbb{Z}, \\ &P \longrightarrow V : s \leftarrow k + x \cdot e \pmod{q}. \end{aligned}$$

Victor now verifies that Peggy knows the secret discrete logarithm x by verifying that $r = g^s \cdot y^{-e}$. Let us examine this protocol in more detail.

Completeness: We first note that the protocol is complete, in that if Peggy actually knows the discrete logarithm then Victor will accept the protocol since

$$g^s \cdot y^{-e} = g^{k+x \cdot e} \cdot (g^x)^{-e} = g^k = r.$$

Soundness: If Peggy does not know the discrete logarithm x then one can informally argue that she will only be able to cheat with probability $1/q$, which is much better than the $1/2$ from the earlier graph-isomorphism-based protocol. We can however show that the protocol has something called the special soundness property.

Definition 21.3 (Special Soundness)**.** *Suppose that we have two protocol runs with transcripts* (r, e, s) *and* (r, e', s')*. Note that the commitments are equal but that the challenges (and hence responses) are different. A protocol is said to have the special soundness property if given two such transcripts we can recover* x*.*

As an example, for our Schnorr protocol above, we have that given two such verifying transcripts we have that

$$r = g^s \cdot y^{-e} = g^{s'} \cdot y^{-e'} = r.$$

This implies in turn that

$$s + x \cdot (-e) = s' + x \cdot (-e') \pmod{q}.$$

Hence, we recover x via

$$x \leftarrow \frac{s - s'}{e - e'} \pmod{q}.$$

Notice that this proof of soundness is almost exactly the same as our use of the forking lemma to show that Schnorr signatures are EUF-CMA secure assuming discrete logarithms are hard. The above algorithm, which takes (r, e, s) and (r, e', s') and outputs the discrete logarithm x, is a knowledge extractor as defined below. It is the existence of this algorithm which shows we have a zero-knowledge proof of knowledge, as defined below, and not just a zero-knowledge proof.

More formally, suppose we have a statement, say $X \in \mathcal{L}$, where $\mathcal{L}$ is some language in $\mathcal{NP}$. Since the language lies in $\mathcal{NP}$ we know there exists a witness w. Now a zero-knowledge proof is an interactive protocol which given a statement $X \in \mathcal{L}$ will allow *an infinitely powerful prover* to demonstrate that $X \in \mathcal{L}$. Note here that the prover may not actually "know" the witness. However, a protocol is said to be a zero-knowledge proof of knowledge if it is a zero-knowledge proof and there exists an algorithm, called a knowledge extractor E, which can use a valid prover to output the witness w.

In our above example we take the prover and run her once, then rewind her to the point when she asks for the verifier's challenge, we then supply her with another challenge and thus end up obtaining the two tuples (r, e, s) and (r, e', s'). Then the special soundness implies that there is a knowledge extractor E which takes as input (r, e, s) and (r, e', s') and outputs the witness. We write $x \leftarrow E\left((r, e, s), (r, e', s')\right)$.

Zero-Knowledge: But does Victor learn anything from the protocol? The answer to this is no, since Victor could simulate the whole transcript in the following way.

- $e \leftarrow \mathbb{Z}/q\mathbb{Z}$.
- $r \leftarrow g^s \cdot y^{-e}$.
- Output the transcript

$$\begin{aligned} P &\longrightarrow V : r, \\ V &\longrightarrow P : e, \\ P &\longrightarrow V : s. \end{aligned}$$

In other words the protocol is zero-knowledge, in that someone cannot tell the simulation of a transcript from a real transcript. This is exactly the same simulation we used when simulating the signing queries in our proof of security of Schnorr signatures in Theorem 16.12.

21.3.2. Formalizing Sigma Protocols: Before we discuss other Sigma protocols, we introduce some notation to aid our discussion. We assume we have some statement $X \in \mathcal{L}$, for some $\mathcal{NP}$ language $\mathcal{L}$, and we want to prove that the prover "knows" the underlying witness w.

Suppose we wish to prove knowledge of the variable x via a Sigma protocol, then

- $R(w, k)$ denotes the algorithm used to compute the commitment r, where k is the random value used to produce the commitment.
- e is the challenge from a set $\mathbb{E}$.
- $S(e, w, k)$ denotes the algorithm which the prover uses to compute her response $s \in \mathbb{S}$ given e.
- $V(r, e, s)$ denotes the verification algorithm.
- $S'(e, s)$ denotes the simulator's algorithm which creates a value of a commitment r which will verify the transcript (r, e, s), for random input values $e \in \mathbb{E}$ and s.
- $E\left((r, e, s), (r, e', s')\right)$ is the knowledge extractor which will output w.

All algorithms are assumed to implicitly have as input the public statement X for which w is a witness. Using this notation Schnorr's identification protocol becomes the following. The statement X is that

$$\exists x \text{ such that } y = g^x,$$

and the witness is $w = x$. We then have

$$\begin{aligned} R(x, k) &:= r \leftarrow g^k, \\ S(e, x, k) &:= s \leftarrow k + e \cdot x \pmod{q}, \\ V(r, e, s) &:= \textbf{true} \text{ if and only if } (r = g^s \cdot y^{-e}), \\ S'(e, s) &:= r \leftarrow g^s \cdot y^{-e}, \\ E((r, e, s), (r, e', s')) &:= x \leftarrow \frac{s - s'}{e - e'} \pmod{q}. \end{aligned}$$

21.3.3. Associated Identification Protocol: We can create an identification protocol from *any* Sigma protocol as follows: We have some statement X which is bound to an entity P such that P has been given the witness w for the statement being valid. To prove that P really does know w without revealing anything about w, we execute the following protocol:

$$\begin{aligned} P &\longrightarrow V : r \leftarrow R(w, k), \\ V &\longrightarrow P : e \leftarrow \mathbb{E}, \\ P &\longrightarrow V : s \leftarrow S(e, w, k), \end{aligned}$$

where the verifier accepts the claimed identity if and only if $V(r, e, s)$ returns **true**. One can think of w in this protocol as a very strong form of "password" authentication, where no information about the "password" is leaked. Of course this will only be secure if finding the witness given only the statement is a hard problem, since otherwise anyone could compute the witness on their own.

21.3.4. Associated Signature Schemes: We have already seen how the Schnorr signature scheme and Schnorr's identification protocol are related. It turns out that *any* Sigma protocol with the special soundness property can be turned into a digital signature scheme. The method of transformation is called the Fiat–Shamir heuristic, since it only produces a heuristically secure scheme as the security proof requires the use of the random oracle H whose codomain is the set $\mathbb{E}$.

- **Key Generation:** The public key is the statement $X \in \mathcal{L}$, and the secret key is the witness w.

- **Signing:** The signature on a message m is generated by

$$r \leftarrow R(w, k),$$
$$e \leftarrow H(r \| m),$$
$$s \leftarrow S(e, w, k).$$

  Output (e, s) as the signature.
- **Verification:** Generate $r' \leftarrow S'(e, s)$, and then check whether $e = H(r' \| m)$.

Using the same technique as in the proof of Theorem 16.12 we can prove the following.

Theorem 21.4. *In the random oracle model let A denote an* EUF-CMA *adversary against the above signature scheme with advantage ϵ, making q_H queries to its hash function H. Then there is an adversary B against the associated Sigma protocol which given $X \in \mathcal{L}$ can extract the witness w with advantage ϵ' such that*

$$\epsilon' \geq \frac{\epsilon^2}{q_H} - \frac{\epsilon}{q_H}.$$

PROOF. The proof is virtually identical to that of Theorem 16.12. We take the adversary A and wrap it inside another algorithm A' which does not make queries to a signature oracle. This is done using the simulator S' for the Sigma protocol. We then apply the forking lemma to A' in order to construct an algorithm B which will output a pair of tuples (r, e, s) and (r, e', s'). We then pass these tuples to the knowledge extractor E in order to recover the witness w. □

21.3.5. Non-interactive Proofs: Sometimes we want to prove something in zero-knowledge but not have the interactive nature of a protocol. For example one entity may be sending some encrypted data to another entity, but wants to prove to anyone seeing the ciphertext that it encrypts a value from a given subset. If a statement can be proved with a Sigma protocol we can turn it into a non-interactive proof by replacing the verifier's challenge component with a hash of the *commitment and the statement*. This last point is often forgotten, since it is not needed in the signature example above, and hence people forget about it when producing general non-interactive proofs. Hence, in the Schnorr proof of knowledge of discrete logarithms protocol we would define the challenge as

$$e \leftarrow H(r \| g \| y).$$

21.3.6. Chaum–Pedersen Protocol: We now present a Sigma protocol called the Chaum–Pedersen protocol which was first presented in the context of electronic cash systems, but which has very wide application. Suppose Peggy wishes to prove she knows two discrete logarithms

$$y_1 = g^{x_1} \text{ and } y_2 = h^{x_2}$$

such that $x_1 = x_2$, i.e. we wish to present not only a proof of knowledge of the discrete logarithms, but also a proof of equality of the hidden discrete logarithms. We assume that g and h generate groups of prime order q, and we denote the common discrete logarithm by x to ease notation. Using our prior notation for Sigma protocols, the Chaum–Pedersen protocol can be expressed via

$$\begin{aligned}
R(x, k) &:= (r_1, r_2) \leftarrow (g^k, h^k), \\
S(e, x, k) &:= s \leftarrow k - e \cdot x \pmod{q}, \\
V((r_1, r_2), e, s) &:= \textbf{true} \text{ if and only if } (r_1 = g^s \cdot y_1^e \text{ and } r_2 = h^s \cdot y_2^e), \\
S'(e, s) &:= (r_1, r_2) \leftarrow (g^s \cdot y_1^e, h^s \cdot y_2^e), \\
E((r, e, s), (r, e', s')) &:= x \leftarrow \frac{s' - s}{e - e'} \pmod{q}.
\end{aligned}$$

Note how this resembles two concurrent runs of the Schnorr protocol, but with a single challenge value. The Chaum–Pedersen protocol is clearly both complete and has the zero-knowledge property,

the second fact follows since the simulation $S'(e, s)$ produces transcripts which are indistinguishable from a real transcript.

We show it is sound, by showing it has the special soundness property. Hence, we assume two protocol runs with the same commitments (r_1, r_2), but with different challenges e and e', and corresponding valid responses s and s'. With this data we need to show that this reveals the common discrete logarithm via the extractor E, *and* that the discrete logarithm is indeed common. Since the two transcripts pass the verification test we have that

$$r_1 = g^s \cdot y_1^e = g^{s'} \cdot y_1^{e'} \text{ and } r_2 = h^s \cdot y_2^e = h^{s'} \cdot y_2^{e'}.$$

But this implies that ${y_1}^{e-e'} = g^{s'-s}$ and ${y_2}^{e-e'} = h^{s'-s}$, and so

$$(e - e') \cdot \text{dlog}_g(y_1) = s' - s \text{ and } (e - e') \cdot \text{dlog}_h(y_2) = s' - s.$$

Hence, the two discrete logarithms are equal and can be extracted from

$$x = \frac{s' - s}{e - e'} \pmod{q}.$$

21.3.7. Proving Knowledge of Pedersen Commitments: Often one commits to a value using a commitment scheme, but the receiver is not willing to proceed unless one proves one knows the value committed to. In other words the receiver will only proceed if he knows that the sender will at some point *be able* to reveal the value committed to. For the commitment scheme

$$B(x) = g^x$$

this is simple, we simply execute Schnorr's protocol for proof of knowledge of a discrete logarithm. For Pedersen commitments $B_a(x) = h^x \cdot g^a$ we need something different. In essence we wish to prove knowledge of x_1 and x_2 such that

$$y = {g_1}^{x_1} \cdot {g_2}^{x_2}$$

where g_1 and g_2 are elements in a group of prime order q. We note that the following protocol generalizes easily to the case when we have more bases, i.e.

$$y = {g_1}^{x_1} \cdots {g_n}^{x_n},$$

a generalization that we leave as an exercise. In terms of our standard notation for Sigma protocols we have

$$\begin{aligned}
R(x, (k_1, k_2)) &:= r \leftarrow {g_1}^{k_1} \cdot {g_2}^{k_2}, \\
S(e, (x_1, x_2), (k_1, k_2)) &:= (s_1, s_2) \leftarrow (k_1 + e \cdot x_1 \pmod{q}, k_2 + e \cdot x_2 \pmod{q}), \\
V(r, e, (s_1, s_2)) &:= \textbf{true} \text{ if and only if } (g_1^{s_1} \cdot g_2^{s_2} = y^e \cdot r), \\
S'(e, (s_1, s_2)) &:= r \leftarrow g_1^{s_1} \cdot g_2^{s_2} \cdot y^{-e}.
\end{aligned}$$

We leave it to the reader to verify that this protocol is complete and zero-knowledge, and to work out the knowledge extractor.

21.3.8. "Or" Proofs: Sometimes the statement about which we wish to execute a Sigma protocol is not as clear cut as the previous examples. For example, suppose we wish to show we know either a secret x or a secret y, without revealing which of the two secrets we know. This is a very common occurrence which arises in a number of advanced protocols, including the voting protocol we consider later in this chapter. It turns out that to show knowledge of one thing or another can be performed using an elegant protocol due to Cramer, Damgård and Schoenmakers.

First assume that there already exist Sigma protocols to prove knowledge of both secrets individually. We will combine these two Sigma protocols together into one protocol which proves the statement we require. The key idea is as follows: For the secret we know, we run the Sigma

protocol as normal, however, for the secret we do not know, we run the simulated Sigma protocol. These two protocols are then linked together by linking the commitments.

As a high-level example, suppose the Sigma protocol for proving knowledge of x is given by the set of algorithms

$$R_1(x, k_1), S_1(e_1, x, k_1), V_1(r_1, e_1, s_1), S_1'(e_1, s_1), E_1((r_1, e_1, s_1), (r_1, e_1', s_1')).$$

Similarly we let the Sigma protocol to prove knowledge of y be given by

$$R_2(y, k_2), S_2(e_2, y, k_2), V_2(r_2, e_2, s_2), S_2'(e_2, s_2), E_2((r_2, e_2, s_2), (r_2, e_2', s_2')).$$

We assume in what follows that the challenges e_1 and e_2 are bit strings of the same length. What is important is that they come from the same set $\mathbb{E}$, and can be combined in a one-time-pad-like manner.[4]

Now suppose we know x, but not y, then our algorithms for the combined proof become:

$$\begin{aligned}
R(x, k_1) &:= (r_1, r_2) \leftarrow \begin{cases} r_1 \leftarrow R_1(x, k_1) \\ e_2 \leftarrow \mathbb{E} \\ s_2 \leftarrow \mathbb{S}_2 \\ r_2 \leftarrow S_2'(e_2, s_2), \end{cases} \\
S(e, x, k_1) &:= (e_1, e_2, s_1, s_2) \leftarrow \begin{cases} e_1 \leftarrow e \oplus e_2 \\ s_1 \leftarrow S_1(e_1, x, k_1), \end{cases} \\
V((r_1, r_2), e, (e_1, e_2, s_1, s_2)) &:= \textbf{true} \text{ if and only if} \\
&\qquad e = e_1 \oplus e_2 \\
&\qquad \text{and } V_1(r_1, e_1, s_1) \\
&\qquad \text{and } V_2(r_2, e_2, s_2), \\
S'(e, (e_1, e_2, s_1, s_2)) &:= (r_1, r_2) \leftarrow \big(S_1'(e_1, s_1), S_2'(e_2, s_2).\big)
\end{aligned}$$

Note that the prover does not reveal the value of e_1, e_2 or s_2 until the response stage of the protocol. Also note that in the simulated protocol the correct distributions of e, e_1 and e_2 are such that $e = e_1 \oplus e_2$. The protocol for the case where we know y but not x follows by reversing the roles of e_1 and e_2, and r_1 and r_2 in the algorithms R, S and V. If the prover knows both x and y then they can execute either of the two possibilties. The completeness, soundness and zero-knowledge properties follow from the corresponding properties of the original Sigma protocols.

These "Or" proofs can be extended to an arbitrary number of disjunctions of statements in the obvious manner: Given n statements of which the prover only knows one secret,

- Simulate $n - 1$ statements using the simulations and challenges e_i
- Commit as usual to the known statement
- Generate a correct challenge for the known statement via

$$e = e_1 \oplus \cdots \oplus e_n.$$

Example 1: We now present a simple example which uses the Schnorr protocol as a building block. Suppose we wish to prove knowledge of either x_1 or x_2 such that $y_1 = g^{x_1}$ and $y_2 = g^{x_2}$, where g lies in a group G of prime order q. We assume that the prover knows x_i but not x_j where $i \neq j$.

The prover's commitment, (r_1, r_2), is computed by selecting e_j and k_i uniformly at random from ${\mathbb{F}_q}^*$ and s_j uniformly at random from G. They then compute $r_i \leftarrow g^{k_i}$ and $r_j \leftarrow g^{s_j} \cdot y_j^{-e_j}$.

[4] So for example we could use addition if they came from a finite field.

On receiving the challenge $e \in \mathbb{F}_q{}^*$ the prover computes

$$\begin{aligned} e_i &\leftarrow e - e_j \pmod q \\ s_i &\leftarrow k_i + e_i \cdot x_i \pmod q. \end{aligned}$$

Note that we have replaced $\oplus$ in computing the "challenge" e_i with addition modulo q; a moment's thought reveals that this is a better way to preserve the relative distributions in this example since arithmetic associated with the challenge is performed in $\mathbb{F}_q$. The prover then outputs (e_1, e_2, s_1, s_2). The verifier checks the proof by checking that $e = e_1 + e_2 \pmod q$ and $r_1 = g^{s_1} \cdot y_1^{-e_1}$ and $r_2 = g^{s_2} \cdot y_2^{-e_2}$.

Example 2: We end this section by giving a protocol which will be required when we discuss voting schemes. It is obtained by combining the protocol for proving knowledge of Pedersen commitments with "Or" proofs. Consider the earlier commitment scheme given by

$$B_a(x) \leftarrow h^x g^a,$$

where $G = \langle g \rangle$ is a finite abelian group of prime order q, h is an element of G whose discrete logarithm with respect to g is unknown, x is the value being committed to, and a is a random value. We are interested in the case where the value committed to is restricted to be either plus or minus one, i.e. $x \in \{-1, 1\}$. It will be important in our application for the person committing to prove that their commitment is from the set $\{-1, 1\}$ without revealing what the actual value of the committed value is. To do this we execute the following protocol.

- As well as publishing the commitment $B_a(x)$, Peggy also chooses random numbers d, r and w modulo q and then publishes α_1 and α_2 where

$$\alpha_1 \leftarrow \begin{cases} g^r \cdot (B_a(x) \cdot h)^{-d} & \text{if } x = 1 \\ g^w & \text{if } x = -1, \end{cases}$$

$$\alpha_2 \leftarrow \begin{cases} g^w & \text{if } x = 1 \\ g^r \cdot \left(B_a(x) \cdot h^{-1}\right)^{-d} & \text{if } x = -1. \end{cases}$$

- Victor now sends a random challenge e to Peggy.
- Peggy responds by setting

$$\begin{aligned} d' &\leftarrow e - d, \\ r' &\leftarrow w + a \cdot d'. \end{aligned}$$

  Then Peggy returns the values

$$(e_1, e_2, r_1, r_2) \leftarrow \begin{cases} (d, d', r, r') & \text{if } x = 1 \\ (d', d, r', r) & \text{if } x = -1. \end{cases}$$

- Victor then verifies that the following three equations hold;

$$\begin{aligned} e &= e_1 + e_2, \\ g^{r_1} &= \alpha_1 \cdot (B_a(x) \cdot h)^{e_1}, \\ g^{r_2} &= \alpha_2 \cdot (B_a(x) \cdot h^{-1})^{e_2}. \end{aligned}$$

To show that the above protocol works we need to show that

(1) If Peggy responds honestly then Victor will verify that the above three equations hold.
(2) If Peggy has not committed to plus or minus one then she will find it hard to produce a response to Victor's challenge which is correct.
(3) The protocol reveals no information to any party as to the exact value of Peggy's commitment, bar that it comes from the set $\{-1, 1\}$.

We leave the verification of these three points to the reader. Note that the above protocol can clearly be conducted in a non-interactive manner by defining $e = H(\alpha_1 \| \alpha_2 \| B_a(x))$.

21.4. An Electronic Voting System

In this section we describe an electronic voting system which utilizes some of the primitives we have been discussing in this chapter and in earlier chapters. In particular we make use of secret sharing schemes from Chapter 19, commitment schemes from Chapter 20, and zero-knowledge proofs from this chapter. The purpose is to show how basic cryptographic primitives can be combined into a complicated application giving real value. One can consider an electronic voting scheme to be a special form of secure multi-party computation, a topic which we shall return to in Chapter 22.

Our voting system will assume that we have m voters, and that there are n centres which perform the tallying. The use of a multitude of tallying centres is to allow voter anonymity and stop a few centres colluding to fix the vote. We shall assume that voters are only given a choice of two candidates, for example Democrat or Republican.

The voting system we shall describe will have the following seven properties.

(1) Only authorized voters will be able to vote.
(2) No one will be able to vote more than once.
(3) No stakeholder will be able to determine how someone else has voted.
(4) No one can duplicate someone else's vote.
(5) The final result will be correctly computed.
(6) All stakeholders will be able to verify that the result was computed correctly.
(7) The protocol will work even in the presence of some bad parties.

System Set-up: Each of the n tally centres has a public key encryption function E_i. We assume a finite abelian group G is fixed, of prime order q, and two elements $g, h \in G$ are selected for which no party (including the tally centres) knows the discrete logarithm $h = g^x$. Each voter has a public key signature algorithm, with which they sign all messages. This last point is to ensure only valid voters vote, and we will ignore this issue in what follows as it is orthogonal to the points we want to bring out.

Vote Casting: Each of the m voters picks a vote v_j from the set $\{-1, 1\}$. The voter picks a random blinding value $a_j \in \mathbb{Z}/q\mathbb{Z}$ and publishes their vote $B_j \leftarrow B_{a_j}(v_j)$, using the Pedersen commitment scheme. This vote is public to all participating parties, both tally centres and other voters. Along with the vote B_j the voter also publishes a non-interactive version of the protocol from Section 21.3.8 to show that the vote was indeed chosen from the set $\{-1, 1\}$. The vote and its proof are then digitally signed using the signing algorithm of the voter.

Vote Distribution: We now need to distribute the votes cast around the tally centres so that the final tally can be computed. To share the a_j and v_j around the tallying centres each voter employs Shamir secret sharing as follows: Each voter picks two random polynomials modulo q of degree $t < n$,

$$R_j(X) \leftarrow v_j + r_{1,j} \cdot X + \cdots + r_{t,j} \cdot X^t,$$
$$S_j(X) \leftarrow a_j + s_{1,j} \cdot X + \cdots + s_{t,j} \cdot X^t.$$

The voter computes

$$(u_{i,j}, w_{i,j}) = (R_j(i), S_j(i)) \text{ for } 1 \leq i \leq n.$$

The voter encrypts the pair $(u_{i,j}, w_{i,j})$ using the ith tally centre's encryption algorithm E_i. This encrypted share is sent to the relevant tally centre. The voter then publishes its commitment to the polynomial $R_j(X)$ by publicly posting $B_{l,j} \leftarrow B_{s_{l,j}}(r_{l,j})$ for $1 \leq l \leq t$, again using the earlier commitment scheme.

Consistency Check: Each centre i needs to check that the values of $(u_{i,j}, w_{i,j})$ it has received from voter j are consistent with the commitment made by the voter. This is done by verifying the following equation:

$$\begin{aligned} B_j \cdot \prod_{\ell=1}^{t} B_{\ell,j}{}^{i^\ell} &= B_{a_j}(v_j) \cdot \prod_{\ell=1}^{t} B_{s_{\ell,j}}(r_{\ell,j})^{i^\ell} \\ &= h^{v_j} \cdot g^{a_j} \cdot \prod_{\ell=1}^{t} (h^{r_{\ell,j}} \cdot g^{s_{\ell,j}})^{i^\ell} \\ &= h^{\left(v_j + \sum_{\ell=1}^{t} r_{\ell,j} \cdot i^\ell\right)} \cdot g^{\left(a_j + \sum_{\ell=1}^{t} s_{\ell,j} \cdot i^\ell\right)} \\ &= h^{u_{i,j}} g^{w_{i,j}}. \end{aligned}$$

Tally Counting: Tally centre i now computes and publicly posts its sum of the shares of the votes cast $T_i = \sum_{j=1}^{m} u_{i,j}$, plus it posts its sum of shares of the blinding factors $A_i = \sum_{j=1}^{m} w_{i,j}$. Every other party, both other centres and voters, can check that this has been done correctly by verifying that

$$\prod_{j=1}^{m} \left(B_j \cdot \prod_{\ell=1}^{t} B_{\ell,j}{}^{j^\ell} \right) = \prod_{j=1}^{m} h^{u_{i,j}} \cdot g^{w_{i,j}} = h^{T_i} \cdot g^{A_i}.$$

Any party can compute the final tally by taking t of the values T_i and interpolating them to reveal the final tally. This is because T_i is the evaluation at i of a polynomial which shares out the sum of the votes. To see this we have

$$\begin{aligned} T_i &= \sum_{j=1}^{m} u_{i,j} = \sum_{j=1}^{m} R_j(i) \\ &= \left(\sum_{j=1}^{m} v_j \right) + \left(\sum_{j=1}^{m} r_{1,j} \right) \cdot i + \cdots + \left(\sum_{j=1}^{m} r_{t,j} \right) \cdot i^t. \end{aligned}$$

If the final tally is negative then the majority of people voted -1, whilst if the final tally is positive then the majority of people voted $+1$. You should now convince yourself that the above protocol has the seven properties we said it would at the beginning.

Chapter Summary

- An interactive proof of knowledge leaks no information if the transcript could be simulated without the need for the secret information.
- Both interactive proofs and zero-knowledge proofs are very powerful constructs; they can be used to prove any statement in $\mathcal{PSPACE}$.
- Interactive proofs of knowledge can be turned into digital signature algorithms by replacing the challenge by the hash of the commitment concatenated with the message.
- Quite complicated protocols can then be built on top of our basic primitives of encryption, signatures, commitment and zero-knowledge proofs. As an example we gave an electronic voting protocol.

Further Reading

The book by Goldreich has more details on zero-knowledge proofs, whilst a good overview of this area is given in the first edition of Stinson's book. The voting scheme we describe is given in the paper of Cramer et al. from Eurocrypt.

R. Cramer, M. Franklin, B. Schoenmakers and M. Yung. *Multi-authority secret-ballot elections with linear work*. In Advances in Cryptology – Eurocrypt 1996, LNCS 1070, 72–83, Springer, 1996.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness*. Springer, 1999.

D. Stinson. *Cryptography: Theory and Practice*. First Edition. CRC Press, 1995.

CHAPTER 22

Secure Multi-party Computation

Chapter Goals

- To introduce the concept of multi-party computation.
- To present a two-party protocol based on Yao's garbled-circuit construction.
- To present a multi-party protocol based on Shamir secret sharing.

22.1. Introduction

Secure multi-party computation is an area of cryptography which deals with two or more parties computing a function on their private inputs. They wish to do so in a way that means that their private inputs still remain private. Of course depending on the function being computed, some information about the inputs may leak. The classical example is the so-called millionaires problem; suppose a bunch of millionaires have a lunch time meeting at an expensive restaurant and decide that the richest of them will pay the bill. However, they do not want to reveal their actual wealth to each other. This is an example of a secure multi-party computation. The inputs are the values x_i, which denote the wealth of each party, and the function to be computed is

$$f(x_1, \ldots, x_n) = i \text{ where } x_i > x_j \text{ for all } i \neq j.$$

Clearly, if we compute such a function, then some information about party i's value leaks; i.e. that it is greater than all the other values. However, we require in secure multi-party computation that this is the only information which leaks; even to the parties participating in the protocol.

One can consider a number of our previous protocols as being examples of secure multi-party computation. For example, the voting protocol given previously involves the computation of the result of each party voting, without anyone learning the vote being cast by a particular party. Encryption is a multi-party computation between three parties: the sender, the receiver and the adversary. Only the sender has an input (which is the message to be encrypted) and only the receiver has an output (which is the message when decrypted). In fact we can essentially see all of cryptography as some form of multi-party computation.

One solution to securely evaluating a function is for all the parties to send their inputs to a trusted third party. This trusted party then computes the function and passes the output back to the parties. However, we want to remove such a trusted third party entirely. Intuitively a multi-party computation is said to be secure if the information which is leaked is precisely that which would have leaked if the computation had been conducted by encrypting messages to a trusted third party.

This is not the only security issue that needs to be addressed when considering secure multi-party computation. There are two basic security models:

- In the first model the parties are guaranteed to follow the protocols, but are interested in breaking the privacy of their fellow participants. Such adversaries are called honest-but-curious, and they in some sense correspond to passive adversaries in other areas of

cryptography. Whilst honest-but-curious adversaries follow the protocol, a number of them could combine their different internal data so as to subvert the security of the non-corrupt parties.

- In the second model the adversaries can deviate from the protocol and may wish to pass incorrect data around so as to subvert the computation of the function. Again we allow such adversaries to talk to each other in a coalition. Such an adversary is called a malicious adversary. In such situations we would like the protocol to still complete, and compute the correct function, i.e. it should be both *correct* and *robust*. In this book we will not discuss modern protocols which trade robustness for other benefits.

There is a problem though. If we assume that communication is asynchronous, which is the most practically relevant situation, then some party must go last. In such a situation one party may have learnt the outcome of the computation, but one party may not have the value yet (namely the party which receives the last message). Any malicious party can clearly subvert the protocol by not sending the last message. Usually malicious adversaries are assumed not to perform such an attack. A protocol which is said to be secure against an adversary which can delete the final message is said to be fair.

In what follows we shall mainly explain the basic ideas behind secure multi-party computation in the case of honest-but-curious adversaries. We shall touch on the case of malicious adversaries for one of our examples though, as it provides a nice example of an application of various properties of Shamir secret sharing.

If we let n denote the number of parties which engage in the protocol, we would like to create protocols for secure multi-party computation which are able to tolerate a large number of corrupt parties. It turns out that there is a theoretical limit on the number of parties whose corruption can be tolerated.

- For the case of honest-but-curious adversaries we can tolerate fewer than $n/2$ corrupt parties, for computationally unbounded adversaries.
- If we restrict ourselves to computationally bounded adversaries then we can tolerate up to $n-1$ corrupt parties in the case of honest-but-curious adversaries.
- However, for a malicious adversary we can tolerate up to $n/3$ corrupt parties if we assume computationally unbounded adversaries.
- If we assume computationally bounded adversaries we can only tolerate less than $n/2$, unless we are prepared to accept an unfair/unrobust protocol in which case we can tolerate up to $n-1$ corrupt parties.

Protocols for secure multi-party computation usually fall into one of two distinct families. The first is based on an idea of Yao called a garbled circuit or Yao circuit: in this case one presents the function to be computed as a binary circuit, and then one "encrypts" the gates of this circuit to form the garbled circuit. This approach is clearly based on a computational assumption, i.e. that the encryption scheme is secure. The second approach is based on secret sharing schemes: here one usually represents the function to be computed as an arithmetic circuit. In this second approach one uses a perfectly secure secret sharing scheme to obtain perfect security.

It turns out that the first approach seems better suited to the case where there are two parties, whilst the second approach seems better suited to the case of three or more parties. In our discussion below we will present a computationally secure solution for the two-party case in the presence of honest-but-curious adversaries, based on Yao circuits. This approach can be extended to more than two parties and a malicious adversary, but doing this is beyond the scope of this book. We then present a protocol for the multi-party case which is perfectly secure. We sketch two versions, one which provides security against honest-but-curious adversaries and one which provides security against malicious adversaries.

22.2. The Two-Party Case

We shall in this section consider the method of secure multi-party computation based on garbled circuits. We suppose there are two parties A and B with inputs x and y respectively, and that A wishes to compute $f_A(x, y)$ and B wishes to compute $f_B(x, y)$. Recall this needs to be done without B learning anything about x or $f_A(x, y)$, except what he can deduce from $f_B(x, y)$ and y, with a similar privacy statement applying to A.

First note that it is enough for B to receive the output of a related function f. To see this we let A have an extra secret input k which is as long as the maximum output of her function $f_A(x, y)$. If we can create a protocol in which B learns the value of the function

$$f(x, y, k) = (k \oplus f_A(x, y), f_B(x, y)),$$

then B simply sends the value of $k \oplus f_A(x, y)$ back to A who can then decrypt it using k, and so determine $f_A(x, y)$. Hence, we will assume that there is only one function which needs to be computed and that its output will be determined by B.

So suppose $f(x, y)$ is the function which is to be computed; we will assume that $f(x, y)$ can be computed in polynomial time. Therefore there is also a polynomial-sized binary circuit which will also compute the output of the function. In the forthcoming example we will write out such a circuit, and so in Figure 22.1 we recall the standard symbols for a binary circuit.

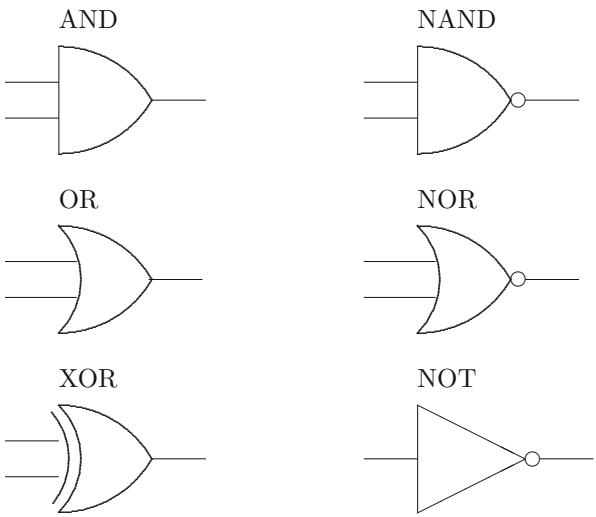


FIGURE 22.1. The basic logic gates

A binary circuit can be represented by a collection of wires $W = \{w_1, \ldots, w_n\}$ and a collection of gates $G = \{g_1, \ldots, g_m\}$. Each gate is a function which takes as input the values of two wires, and produces the value of the output wire. For example suppose g_1 is an AND gate which takes as input wires w_1 and w_2 and produces the output wire w_3. Then gate g_1 can be represented by the following truth table.

w_1	w_2	w_3
0	0	0
0	1	0
1	0	0
1	1	1

In other words, the gate g_i represents a function, $w_3 \leftarrow g_i(w_1, w_2)$ such that

$$0 = g_i(0,0) = g_i(1,0) = g_i(0,1) \text{ and } 1 = g_i(1,1).$$

22.2.1. Garbled Circuit Construction: In Yao's protocol one party constructs a garbled circuit (we shall call this party A), the other party evaluates the garbled circuit (we shall call this party B). The garbled circuit is constructed as follows:

- For each wire w_i two random cryptographic keys are selected, k_i^0 and k_i^1. The first one represents the encryption of the zero value and the second represents the encryption of the one value.
- For each wire a random value $\rho_i \in \{0,1\}$ is chosen. This is used to also encrypt the actual wire value. If the actual wire value is v_i then the encrypted, or "external" value, is given by $e_i = v_i \oplus \rho_i$.
- For each gate we compute a "garbled table" representing the function of the gate on these encrypted values. Suppose g_i is a gate with input wires w_{i_0} and w_{i_1} and output wire w_{i_2}, then the garbled table is the following four values, for some encryption function E:

$$c_{a,b}^{w_{i_2}} = E_{k_{w_{i_0}}^{a\oplus\rho_{i_0}}, k_{w_{i_1}}^{b\oplus\rho_{i_1}}}\left(k_{w_{i_2}}^{o_{a,b}} \| \left(o_{a,b} \oplus \rho_{i_2}\right)\right) \text{ for } a,b \in \{0,1\}.$$

  where $o_{a,b} = g_i(a \oplus \rho_{i_0}, b \oplus \rho_{i_1})$.

We do not consider exactly what encryption function is chosen; such a discussion is slightly beyond the scope of this book. If you want further details then look in the Further Reading section at the end of this chapter, or just assume we take an encryption scheme which is suitably secure.

The above may seem rather confusing so we illustrate the method for constructing the garbled circuit with an example. Suppose A and B each have as input two bits; we shall denote A's input wires by w_1 and w_2, whilst we shall denote B's input wires by w_3 and w_4. Suppose they now wish to engage in a secure multi-party computation so that B learns the value of the function

$$f(\{w_1, w_2\}, \{w_3, w_4\}) = (w_1 \wedge w_3) \vee (w_2 \oplus w_4).$$

A circuit to represent this function is given in Figure 22.2.

In Figure 22.2 we also present the garbled values of each wire and the corresponding garbled tables representing each gate. In this example we have the following values of ρ_i:

$$\rho_1 = \rho_4 = \rho_6 = \rho_7 = 1 \text{ and } \rho_2 = \rho_3 = \rho_5 = 0.$$

Consider the first wire; the two garbled values of the wire are k_1^0 and k_1^1, which represent the 0 and 1 values, respectively. Since $\rho_1 = 1$, the external value of the internal 0 value is 1 and the external value of the internal 1 value is 0. Thus we represent the garbled value of the wire by the pair of pairs

$$(k_1^0 \| 1, k_1^1 \| 0).$$

Now we look at the gates, and in particular consider the first gate. The first gate is an AND gate which takes as input the first and third wires. The first entry in this table corresponds to $a = b = 0$. Now the ρ values for the first and third wires are 1 and 0 respectively. Hence, the first entry in the table corresponds to what should happen if the keys k_1^1 and k_3^0 are seen, since

$$1 = 1 \oplus 0 = \rho_1 \oplus a \text{ and } 0 = 0 \oplus 0 = \rho_3 \oplus b.$$

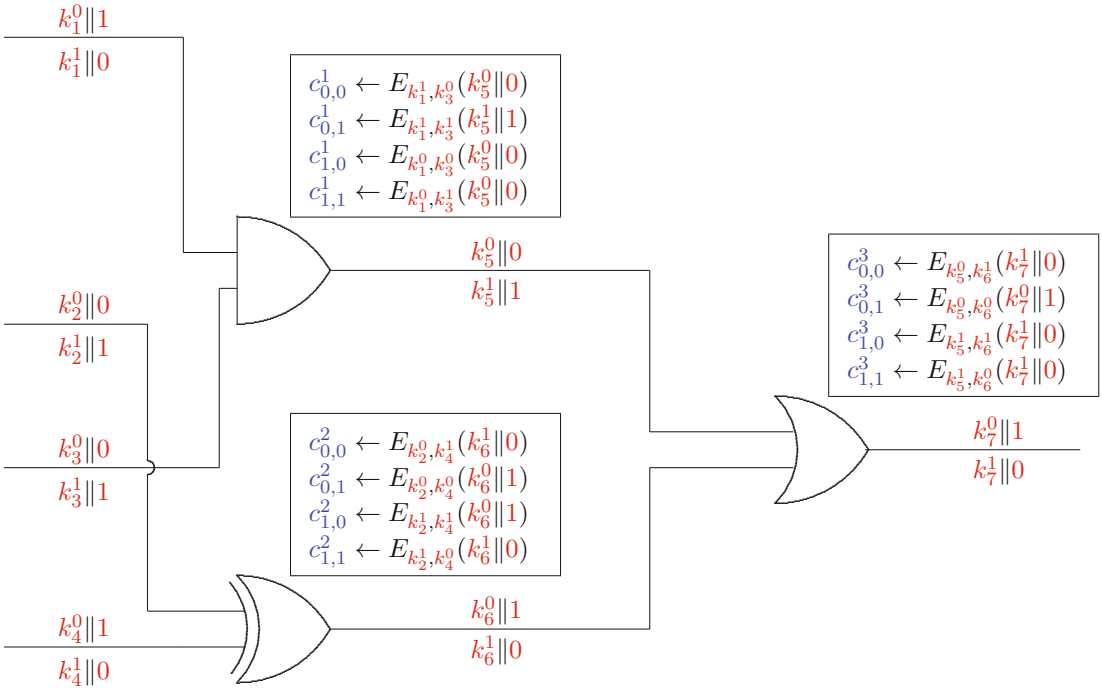


FIGURE 22.2. A garbled circuit

Now the AND gate should produce the 0 output on input of 1 and 0, thus the thing which is encrypted in the first line is the key representing the zero value of the fifth wire, i.e. k_5^0, plus the "external value" of 0, namely $0 = 0 \oplus 0 = 0 \oplus \rho_5$.

22.2.2. Garbled Circuit Evaluation: We now describe how the circuit is evaluated by party B. Please refer to Figure 22.3 for a graphical description of this. We assume that B has obtained in some way the specific garbled values of the input wires marked in blue in Figure 22.3, and the value of ρ_i for the output wires; in our example this is just ρ_7. Firstly party B evaluates the AND gate; he knows that the external value of wire one is 1 and the external value of wire three is 1. Thus he looks up the entry $c_{1,1}^1$ in the table and decrypts it using the two keys he knows, i.e. k_1^0 and k_3^1. He then obtains the value $k_5^0\|0$. He has no idea whether this represents the zero or one value of the fifth wire, since he has no idea as to the value of ρ_5.

Party B then performs the same operation with the exclusive-or gate. This has input wire 2 and wire 4, for which party B knows that the external values are 0 and 1 respectively. Thus party B decrypts the entry $c_{0,1}^2$ to obtain $k_6^0\|1$. A similar procedure is then carried out with the final OR gate, using the keys and external values of the fifth and sixth wires. This results in a decryption which reveals the value $k_7^0\|1$. So the external value of the seventh wire is equal to 1, but party B has been told that $\rho_7 = 1$, and hence the internal value of wire seven will be $0 = 1 \oplus 1$. Hence, the output of the function is the bit 0.

22.2.3. Yao's Protocol: We are now in a position to describe Yao's protocol in detail. The protocol proceeds in five phases as follows:

(1) Party A generates the garbled circuit as above, and transmits to party B only the values $c_{a,b}^i$.

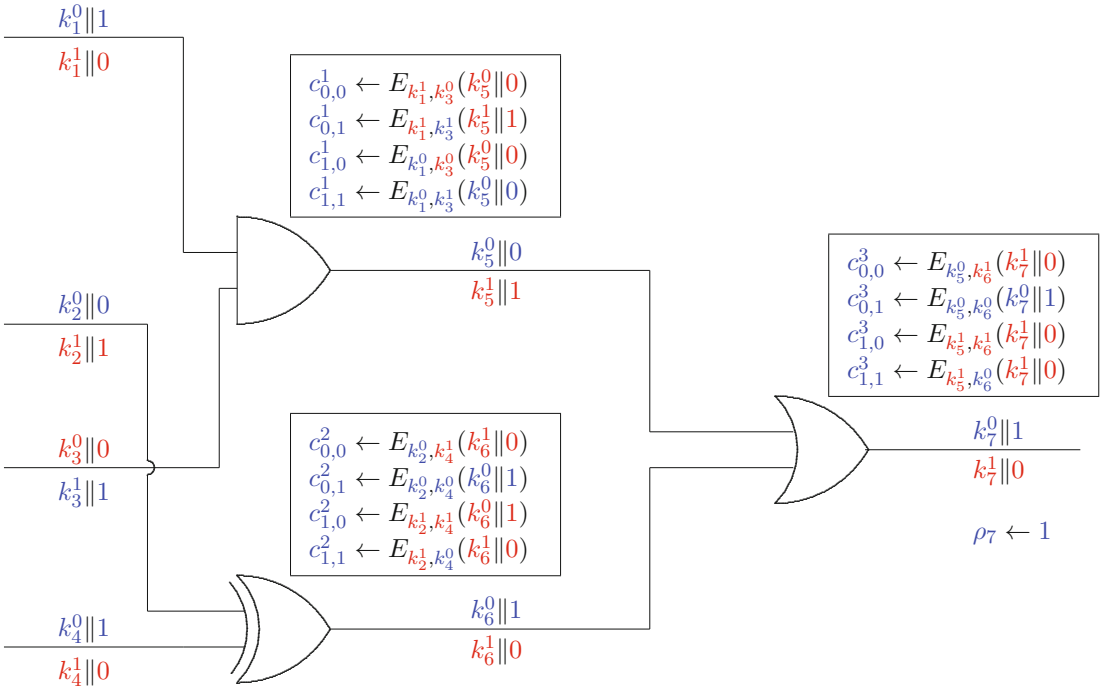


FIGURE 22.3. Evaluating a garbled circuit

(2) Party A then transmits to party B the garbled values of the component of its input wires. For example, suppose that party A's input is $w_1 = 0$ and $w_2 = 0$. Then party A transmits to party B the two values $k_1^0\|1$ and $k_2^0\|0$. Note that party B cannot learn the actual values of w_1 and w_2 from these values since he does not know ρ_1 and ρ_2, and the keys k_1^0 and k_2^0 just look like random keys.

(3) Party A and B then engage in an oblivious transfer protocol, as in Section 20.3, for each of party B's input wires. In our example suppose that party B's input is $w_3 = 1$ and $w_4 = 0$. The two parties execute two oblivious transfer protocols, one with A's input $k_3^0\|0$ and $k_3^1\|1$, and B's input 1, and one with A's input $k_4^0\|1$ and $k_4^1\|0$, and B's input 0. At the end of this oblivious transfer phase party B has learnt $k_3^1\|1$ and $k_4^0\|1$.

(4) Party A then transmits to party B the values of ρ_i for all of the output wires. In our example he reveals the value of $\rho_7 = 1$.

(5) Finally party B evaluates the circuit using the garbled input wire values he has been given, using the technique described above.

In summary, in the first stage all party B knows about the garbled is in the blue items in Figure 22.2, but by the last stage he knows the blue items in Figure 22.3.

In our example we can now assess what party B has learnt from the computation. Party B knows that the output of the final OR gate is zero, which means that the inputs must also be zero, which means that the output of the AND gate is zero and the output of the exclusive-or gate is zero. However, party B already knew that the output of the AND gate will be zero, since his own input was zero. However, party B has learnt that party A's second input wire represented zero, since otherwise the exclusive-or gate would not have output zero. So whilst party A's first input

remains private, the second input does not. This is what we meant by a protocol keeping the inputs private, bar what could be deduced from the output of the function.

22.3. The Multi-party Case: Honest-but-Curious Adversaries

The multi-party case is based on using a secret sharing scheme to evaluate an arithmetic circuit. An arithmetic circuit consists of a finite field $\mathbb{F}_q$ and a polynomial function (which could have many inputs and outputs) defined over the finite field. The idea is that such a function can be evaluated by executing a number of addition and multiplication gates over the finite field.

Given an arithmetic circuit it is clear one could express it as a binary circuit, by simply expanding out the addition and multiplication gates of the arithmetic circuit as their binary circuit equivalents. One can also represent every binary circuit as an arithmetic circuit, since every gate in the binary circuit can be represented as a linear function of the input values to the gate and their products. For example, suppose we represent the binary values 0 and 1 by 0 and 1 in the finite field $\mathbb{F}_q$, and that the characteristic of $\mathbb{F}_q$ is larger than two. We then have that the binary exclusive-or gate can be written as $x \oplus y = -2 \cdot x \cdot y + x + y$ over $\mathbb{F}_q$, and the binary "and" gate can be written as $x \odot y = x \cdot y$.

Whilst the two representations are equivalent it is clear that some functions are easier to represent as binary circuits and some are easier to represent as arithmetic circuits.

As before we shall present the protocol via a running example. We shall suppose we have six parties $P_1, \ldots, P_6$ who have six secret values $x_1, \ldots, x_6$, each of which lie in $\mathbb{F}_p$, for some reasonably large prime p. For example we could take $p \approx 2^{128}$, but in our example to make things easier to represent we will take $p = 101$. The parties are assumed to want to compute the value of the function

$$f(x_1, \ldots, x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \pmod{p}.$$

Hence, the arithmetic circuit for this function consists of three multiplication gates and two addition gates, as in Figure 22.4, where we label the intermediate values as numbered "wires".

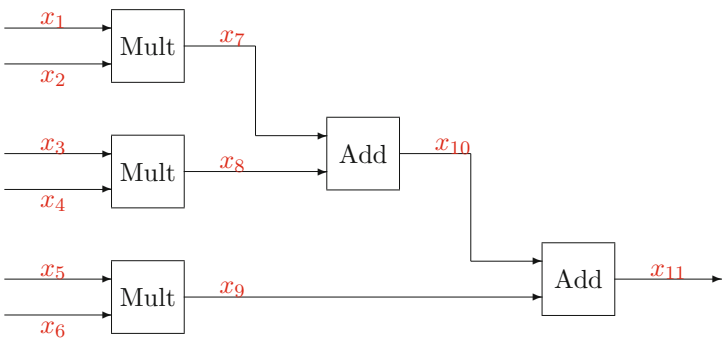


FIGURE 22.4. Graphical representation of the example arithmetic circuit

We will use Shamir's secret sharing, in which case our basic protocol is as follows: The value of each wire x_i is shared between all players, with each player j obtaining a share $x_i^{(j)}$. Clearly, if enough players come together, then they can determine the value of the wire x_i, by the properties of the secret sharing scheme.

Each player can create shares of his or her own input values at the start of the protocol and send a share to each player. Thus we need to show how to obtain the shares of the outputs of a gate, given shares of the inputs of the gate. Recall that in Shamir's secret sharing the shared value

is given by the constant term of a polynomial f of degree t, with the shares being the evaluation of the polynomial at given positions corresponding to each participant $f(i)$.

22.3.1. Addition Gates: First we consider how to compute the `Add` gates. Suppose we have two secrets a and b which are shared using the polynomials

$$f(X) = a + f_1 \cdot X + \cdots + f_t \cdot X^t,$$
$$g(X) = b + g_1 \cdot X + \cdots + g_t \cdot X^t.$$

Each of our parties has a share $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$. Now consider the polynomial

$$h(X) = f(X) + g(X).$$

This polynomial provides a sharing of the sum $c = a + b$, and we have

$$c^{(i)} = h(i) = f(i) + g(i) = a^{(i)} + b^{(i)}.$$

Hence, the parties can compute a sharing of the output of an `Add` gate without any form of communication between them.

22.3.2. Multiplication Gates: Computing the output of a `Mult` gate is more complicated. First we recap the following property of Lagrange interpolation. If $f(X)$ is a polynomial and we distribute the values $f(i)$ then there is a vector $(r_1, \ldots, r_n)$, called the recombination vector, such that

$$f(0) = \sum_{i=1}^{n} r_i \cdot f(i).$$

And the same vector works for all polynomials $f(X)$ of degree at most $n - 1$.

To compute the `Mult` gate we perform the following four steps. We assume as input that each party has a share of a and b via $a^{(i)} = f(i)$ and $b^{(i)} = g(i)$, where $f(0) = a$ and $g(0) = b$. We wish to compute a sharing $c^{(i)} = h(i)$ such that $h(0) = c = a \cdot b$.

- Each party locally computes $d^{(i)} = a^{(i)} \cdot b^{(i)}$.
- Each party produces a polynomial $\delta_i(X)$ of degree at most t such that $\delta_i(0) = d^{(i)}$.
- Each party i distributes to party j the value $d_{i,j} = \delta_i(j)$.
- Each party j computes $c^{(j)} = \sum_{i=1}^{n} r_i \cdot d_{i,j}$.

So why does this work? Consider the first step; here we are actually effectively computing a polynomial $h'(X)$ of degree at most $2 \cdot t$, with $d^{(i)} = h'(i)$, and $c = h'(0)$. Hence, the only problem with the sharing in the first step is that the underlying polynomial has too high a degree. The main thing to note is that if

$$2 \cdot t \leq n - 1 \tag{23}$$

then we have $c = \sum_{i=1}^{n} r_i \cdot d^{(i)}$. Now consider the polynomials $\delta_i(X)$ generated in the second step, and consider what happens when we recombine them using the recombination vector, i.e. set

$$h(X) = \sum_{i=1}^{n} r_i \cdot \delta_i(X).$$

Since the $\delta_i(X)$ are all of degree at most t, the polynomial $h(X)$ is also of degree at most t. We also have that

$$h(0) = \sum_{i=1}^{n} r_i \cdot \delta_i(0) = \sum_{i=1}^{n} r_i \cdot d^{(i)} = c,$$

assuming $2 \cdot t \leq n - 1$. Thus $h(X)$ is a polynomial which *could* be used to share the value of the product. Not only that, but it *is* the polynomial underlying the sharing produced in the final step. To see this notice that

$$h(j) = \sum_{i=1}^{n} r_i \cdot \delta_i(j) = \sum_{i=1}^{n} r_i \cdot d_{i,j} = c^{(j)}.$$

Example: So assuming $t < n/2$ we can produce a protocol which evaluates the arithmetic circuit correctly. We illustrate the method by examining what would happen for our example circuit in Figure 22.4, with $p = 101$. Recall that there are six parties; we shall assume that their inputs are given by

$$x_1 = 20,\ x_2 = 40,\ x_3 = 21,\ x_4 = 31,\ x_5 = 1,\ x_6 = 71.$$

Each party first computes a sharing ${x_i}^{(j)}$ of their secret amongst the six parties. They do this by each choosing a random polynomial of degree $t = 2$ and evaluating it at $j = 1, 2, 3, 4, 5, 6$. The values obtained are then distributed securely to each party. Hence, each party obtains its row of the following table.

	i					
j	1	2	3	4	5	6
1	44	2	96	23	86	83
2	26	0	63	13	52	79
3	4	22	75	62	84	40
4	93	48	98	41	79	10
5	28	35	22	90	37	65
6	64	58	53	49	46	44

As an exercise you should work out the associated polynomials corresponding to each column.

The parties then engage in the multiplication protocol so as to compute sharings of $x_7 = x_1 \cdot x_2$. They first compute their local multiplication, by each multiplying the first two elements in their row of the above table, then they form a sharing of this local multiplication. These sharings of six numbers between six parties are then distributed securely. In our example run each party, for this multiplication, obtains the sharings given by its column of the following table.

	j					
i	1	2	3	4	5	6
1	92	54	20	91	65	43
2	10	46	7	95	7	46
3	64	100	96	52	69	46
4	23	38	41	32	11	79
5	47	97	77	88	29	1
6	95	34	11	26	79	69

Each party then takes the six values obtained and recovers their share of the value of x_7. We find that the six shares of x_7 are given by

$${x_7}^{(1)} = 9,\ {x_7}^{(2)} = 97,\ {x_7}^{(3)} = 54,\ {x_7}^{(4)} = 82,\ {x_7}^{(5)} = 80,\ {x_7}^{(6)} = 48.$$

Repeating the multiplication protocol twice more we also obtain a sharing of x_8 as

$${x_8}^{(1)} = 26,\ {x_8}^{(2)} = 91,\ {x_8}^{(3)} = 38,\ {x_8}^{(4)} = 69,\ {x_8}^{(5)} = 83,\ {x_8}^{(6)} = 80,$$

and x_9 as

$${x_9}^{(1)} = 57,\ {x_9}^{(2)} = 77,\ {x_9}^{(3)} = 30,\ {x_9}^{(4)} = 17,\ {x_9}^{(5)} = 38,\ {x_9}^{(6)} = 93.$$

We are then left with the two addition gates to produce the sharings of the wires x_{10} and x_{11}. These are obtained by locally adding together the various shared values so that

$$x_{11}{}^{(1)} = x_7{}^{(1)} + x_8{}^{(1)} + x_9{}^{(1)} \pmod{101} = 9 + 26 + 57 \pmod{101} = 92,$$

etc. to obtain

$$x_{11}{}^{(1)} = 92,\ x_{11}{}^{(2)} = 63,\ x_{11}{}^{(3)} = 21,\ x_{11}{}^{(4)} = 67,\ x_{11}{}^{(5)} = 100,\ x_{11}{}^{(6)} = 19.$$

The parties then make public these shares, and recover the hidden polynomial, of degree $t = 2$, which produces these sharings, namely $7 + 41 \cdot X + 44 \cdot X^2$. Hence, the result of the multi-party computation is the value 7.

Now assume that more than t parties are corrupt, in the sense that they collude to try to break the privacy of the non-corrupted parties. The corrupt parties can now come together and recover any of the underlying secrets in the scheme, since we have used Shamir secret sharing using polynomials of degree at most t. It can be shown, using the perfect secrecy of the Shamir secret sharing scheme, that as long as no more than t parties are corrupt then the above protocol is perfectly secure.

However, it is only perfectly secure assuming all parties follow the protocol, i.e. we are in the honest-but-curious model. As soon as we allow parties to deviate from the protocol, they can force the honest parties to produce invalid results. To see this just notice that a dishonest party could simply produce an invalid sharing of its product in the second part of the multiplication protocol above.

22.4. The Multi-party Case: Malicious Adversaries

To produce a scheme which is secure against active adversaries either we need to force all parties to follow the protocol or we should be able to recover from errors which malicious parties introduce into the protocol. It is the second of these two approaches which we shall follow in this section, by using the error correction properties of the Shamir secret sharing scheme. As already remarked, the above protocol is not secure against malicious adversaries, due to the ability of an attacker to make the multiplication protocol output an invalid answer. To make the above protocol secure against malicious adversaries we make use of various properties of the Shamir secret sharing scheme.

The protocol runs in two stages: The preprocessing stage does not involve any of the secret inputs of the parties, it depends purely on the number of multiplication gates in the circuit. In the main phase of the protocol the circuit is evaluated as in the previous section, but using a slightly different multiplication protocol. Malicious parties can force the preprocessing stage to fail, however if it completes then the honest parties will be able to evaluate the circuit as required.

The preprocessing phase runs as follows. First, using the techniques from Chapter 19, a pseudo-random secret sharing scheme, PRSS, and a pseudo-random zero sharing scheme, PRZS, are set up. Then for each multiplication gate in the circuit we compute a random triple of sharings $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. This is done as follows:

- Using PRSS generate two random sharings, $a^{(i)}$ and $b^{(i)}$, of degree t.
- Using PRSS generate another random sharing $r^{(i)}$ of degree t.
- Using PRZS generate a sharing $z^{(i)}$, of degree $2 \cdot t$ of zero.
- Each party then locally computes $s^{(i)} = a^{(i)} \cdot b^{(i)} - r^{(i)} + z^{(i)}$. Note that this local computation will produce a degree $2 \cdot t$ sharing of the value $s = a \cdot b - r$.
- Then the players broadcast their values $s^{(i)}$ and try to recover s. Here we make use of the error detection properties of Reed–Solomon codes. If the number of malicious parties is bounded by $t < n/3$, then any error in the degree-$(2 \cdot t)$ sharing will be detected. At this stage the parties abort the protocol if any error is found.
- Now the players locally compute the shares $c^{(i)}$ from $c^{(i)} = s + r^{(i)}$.

Assuming the above preprocessing phase completes successfully, all we need to do is specify how the parties implement a `Mult` in the presence of malicious adversaries. We assume the inputs to the multiplication gate are given by $x^{(i)}$ and $y^{(i)}$ and we wish to compute a sharing $z^{(i)}$ of the product $z = x \cdot y$. From the preprocessing stage, the parties also have for each gate, a triple of shares $a^{(i)}$, $b^{(i)}$ and $c^{(i)}$ such that $c = a \cdot b$. The protocol for the multiplication is then as follows:

- Compute locally, and then broadcast, the values $d^{(i)} = x^{(i)} - a^{(i)}$ and $e^{(i)} = y^{(i)} - b^{(i)}$.
- Reconstruct the values of $d = x - a$ and $e = y - b$.
- Locally compute the shares $z^{(i)} = d \cdot e + d \cdot b^{(i)} + e \cdot a^{(i)} + c^{(i)}$.

Note that the reconstruction in the second step can be completed, even if the corrupt parties transmit invalid values, as long as there are at most $t < n/3$ malicious parties. Due to the error correction properties of Reed–Solomon codes, we can recover from any errors introduced by malicious parties. The above protocol produces a valid sharing of the output of the multiplication gate because

$$\begin{aligned} d \cdot e + d \cdot b + e \cdot a + c &= (x - a) \cdot (y - b) + (x - a) \cdot b + (y - b) \cdot a + c \\ &= ((x - a) + a) \cdot ((y - b) + b) \\ &= x \cdot y = z. \end{aligned}$$

Chapter Summary

- We have explained how to perform two-party secure computation, in the case of honest-but-curious adversaries, using Yao's garbled-circuit construction.
- For the multi-party case we have presented a protocol based on evaluating arithmetic, as opposed to binary, circuits which is based on Shamir secret sharing.
- The main issue with this latter protocol is how to evaluate the multiplication gates. We presented two methods: The first, simpler, method is applicable when one is only dealing with honest-but-curious adversaries, the second, more involved, method is for the case of malicious adversaries.

Further Reading

The original presentation of Yao's idea was apparently given in the talk which accompanied the paper in FOCS 1986, however the paper contains no explicit details of the protocol. It can be transformed into a scheme for malicious adversaries using a general technique of Goldreich et al. The discussion of the secret sharing based solution for the honest and malicious cases closely follows the treatment in Damgård et al.

I. Damgård, M. Geisler, M. Krøigaard and J.B. Nielsen. *Asynchronous multiparty computation: Theory and implementation* In Public Key Cryptography – PKC 2009, LNCS 5443, 160–179, Springer, 2009.

O. Goldreich, S. Micali and A. Wigderson. *How to play any mental game.* In Symposium on Theory of Computing – STOC 1987, 218–229, ACM, 1987.

A.C. Yao. *How to generate and exchange secrets.* In Foundations of Computer Science – FOCS 1986, 162–167, IEEE, 1986.

Appendix

Basic Mathematical Terminology

This appendix is presented as a series of notes which summarize most of the mathematical terminology needed in this book. We present the material in a more formal manner than we did in Chapter 1 and the rest of the book.

A.1. Sets

Here we recap some basic definitions etc. which we list here for completeness.

Definition 100.1 (Set Union, Intersection, Difference and Cartesian Product)**.** *For two sets A, B we define the union, intersection, difference and Cartesian product by*

$$A \cup B = \{x : x \in A \text{ or } x \in B\},$$
$$A \cap B = \{x : x \in A \text{ and } x \in B\},$$
$$A \setminus B = \{x : x \in A \text{ and } x \notin B\},$$
$$A \times B = \{(x, y) : x \in A \text{ and } y \in B\}.$$

The statement $A \subseteq B$ means that for all $x \in A$ it follows that $x \in B$.

Using these definitions one can prove in a standard way all the basic results of set theory that one shows in school using Venn diagrams.

Lemma 100.2. *If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$.*

Proof. Let x be an element of A; we wish to show that x is an element of C. Now as $A \subseteq B$ we have that $x \in B$, and as $B \subseteq C$ we then deduce that $x \in C$. □

Notice that this is a proof whereas an argument using Venn diagrams to demonstrate something is not a proof. Using Venn diagrams to show something merely shows you were not clever enough to come up with a picture which proved the result false.

There are some standard sets which will be of interest in our discussions: $\mathbb{N}$ the set of natural numbers, $\{0, 1, 2, 3, 4, \ldots\}$; $\mathbb{Z}$ the set of integers, $\{0, \pm 1, \pm 2, \pm 3, \pm 4, \ldots\}$; $\mathbb{Q}$ the set of rational numbers, $\{p/q : p \in \mathbb{Z}, q \in \mathbb{N} \setminus \{0\}\}$; $\mathbb{R}$ the set of real numbers; $\mathbb{C}$ the set of complex numbers.

A.2. Relations

Next we define relations and some properties that they have. Relations, especially equivalence relations, play an important part in algebra and it is worth considering them at this stage so it is easier to understand what is going on later.

Definition 100.3 (Relation)**.** *A (binary) relation on a set A is a subset of the Cartesian product $A \times A$.*

This we explain with an example: Consider the relationship "less than or equal to" between natural numbers. This obviously gives us the set

$$\mathsf{LE} = \{(x, y) : x, y \in \mathbb{N},\ x \text{ is less than or equal to } y\}.$$

© Springer International Publishing Switzerland 2016

In much the same way every relationship that you have met before can be written in this set-theoretic way. An even better way to put the above is to define the relation "less than or equal to" to be the set

$$\mathsf{LE} = \{(x, y) : x, y \in \mathbb{N},\ x - y \notin \mathbb{N} \setminus \{0\}\}.$$

Obviously this is a very cumbersome notation so for a relation R on a set S we write

$$x\ R\ y$$

if $(x, y) \in R$, i.e. if we now write $\leq$ for LE we obtain the usual notation $1 \leq 2$ etc. Relations which are of interest in mathematics usually satisfy one or more of the following four properties.

Definition 100.4 (Properties of Relations)**.**

- *A relation R on a set S is reflexive if for all $x \in S$ we have $(x, x) \in R$.*
- *A relation R on a set S is symmetric if $(x, y) \in R$ implies that $(y, x) \in R$.*
- *A relation R on a set S is anti-symmetric if $(x, y) \in R$ and $(y, x) \in R$ implies that $x = y$.*
- *A relation R on a set S is transitive if $(x, y) \in R$ and $(y, z) \in R$ implies that $(x, z) \in R$.*

We return to our example of $\leq$. This relation $\leq$ is certainly reflexive as $x \leq x$ for all $x \in \mathbb{N}$. It is not symmetric as $x \leq y$ does not imply that $y \leq x$, however it is anti-symmetric as $x \leq y$ and $y \leq x$ imply that $x = y$. You should note that it is transitive as well.

Relations like $\leq$ occur so frequently that we give them a name.

Definition 100.5 (Partial Order Relation)**.** *A relation which is a partial order relation if it is reflexive, transitive and anti-symmetric.*

Definition 100.6 (Total Order Relation)**.** *A relation which is transitive and anti-symmetric and for which for all x and y, with $x \neq y$, we have either $(x, y) \in R$ or $(y, x) \in R$ is called a total order relation.*

Whilst every total order relation is a partial order relation, the converse is not true. For example consider the relation of

$$\mathsf{div} = \{(x, y) : x, y \in \mathbb{N}, x \text{ divides } y\}.$$

This is clearly a partial ordering, since

- It is reflexive, as x divides x.
- It is transitive, as x divides y and y divides z implies x divides z.
- It is anti-symmetric, as if x divides y and y divides x then $x = y$.

But it is clearly not a total order as 3 does not divide 4 and 4 does not divide 3.

Another important type of relationship is that of an equivalence relation.

Definition 100.7 (Equivalence Relation)**.** *A relation which is reflexive, symmetric and transitive is called an equivalence relation.*

The obvious example of $\mathbb{N}$ and the relation "is equal to" is an equivalence relation and hence gives this type of relation its name. One of the major problems in any science is that of classification of sets of objects. This amounts to placing the objects into mutually disjoint subsets. An equivalence relation allows us to place elements into disjoint subsets. Each of these subsets is called an equivalence class. If the properties we are interested in are constant over each equivalence class then we may as well restrict our attention to the equivalence classes themselves. This often leads to greater understanding. In the jargon this process is called factoring out by the equivalence relation. It occurs frequently in algebra to define new objects from old, e.g. quotient groups. The following example is probably the most familiar; being a description of modular arithmetic.

Let m be a fixed positive integer. Consider the equivalence relation on $\mathbb{Z}$ which says x is related to y if $(x - y)$ is divisible by m. This is an equivalence relation, which you should check. The

equivalence classes we denote by

$$\begin{aligned}\overline{0} &= \{\ldots, -2\cdot m, -m, 0, m, 2\cdot m, \ldots\},\\ \overline{1} &= \{\ldots, -2\cdot m+1, -m+1, 1, m+1, 2\cdot m+1, \ldots\},\\ \ldots &\quad \ldots\\ \overline{m-1} &= \{\ldots, -m-1, -1, m-1, 2\cdot m-1, 3\cdot m-1, \ldots\}.\end{aligned}$$

Note that there are m distinct equivalence classes, one for each of the possible remainders on division by m. The classes are often called the residue classes modulo m. The resulting set $\{\overline{0}, \ldots, \overline{m-1}\}$ is often denoted by $\mathbb{Z}/m\mathbb{Z}$ as we have divided out by all multiples of m. If m is a prime number, say p, then the resulting set is often denoted $\mathbb{F}_p$ as the resulting object is a field.

A.3. Functions

We give two definitions of functions; the first is wordy and is easier to get hold of, the second is set-theoretic.

Definition 100.8 (Function – v1)**.** *A function is a rule which maps the elements of one set, the domain, to those of another, the codomain. Each element in the domain must map to one and only one element in the codomain (a.k.a. the range of the function).*

The point here is that the function is not just the rule, e.g. $f(x) = x^2$, but also the two sets that one is using. A few examples will suffice.

(1) The rule $f(x) = \sqrt{x}$ is not a function from $\mathbb{R}$ to $\mathbb{R}$ since the square root of a negative number is not in $\mathbb{R}$. It is also not a function (depending on how you define the $\sqrt{}$ symbol) from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}$ since every element of the domain has two square roots in the codomain. But it is a function from $\mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0}$.
(2) The rule $f(x) = 1/x$ is not a function from $\mathbb{R}$ to $\mathbb{R}$ but it is a function from $\mathbb{R} \setminus \{0\}$ to $\mathbb{R}$.
(3) Note that not every element of the codomain need have an element mapping to it. Hence, the rule $f(x) = x^2$ taking elements of $\mathbb{R}$ to elements of $\mathbb{R}$ is a function.

Our definition of a function is unsatisfactory as it would also require a definition of what a rule is. In keeping with the spirit of everything else we have done we give a set-theoretic description.

Definition 100.9 (Function – v2)**.** *A function from the set A to the set B is a subset F of $A \times B$ such that:*

(1) *If $(x, y) \in F$ and $(x, z) \in F$ then $y = z$.*
(2) *For all $x \in A$ there exists a $y \in B$ such that $(x, y) \in F$.*

The set A is called the domain, the set B the codomain. The first condition means that each element in the domain maps to at most one element in the codomain. The second condition means that each element of the domain maps to at least one element in the codomain. Given a function f from A to B and an element x of A then we denote by $f(x)$ the unique element in B such that $(x, f(x)) \in f$.

Composition of Functions: One can compose functions, if the definitions make sense. Say one has a function f from A to B and a function g from B to C, then the function $g \circ f$ is the function with domain A and codomain C consisting of the elements $(x, g(f(x)))$.

Lemma 100.10. *Let f be a function from A to B, let g be a function from B to C and let h be a function from C to D, then we have*

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Proof. Let (a, d) belong to $(h \circ g) \circ f$. Then there exists an $(a, b) \in f$ and a $(b, d) \in (h \circ g)$ for some $b \in B$, by definition of composition of functions. Again by definition there exists a $c \in C$

such that $(b,c) \in g$ and $(c,d) \in h$. Hence $(a,c) \in (g \circ f)$, which shows $(a,d) \in h \circ (g \circ f)$. Hence

$$(h \circ g) \circ f \subseteq h \circ (g \circ f).$$

Similarly one can show the other inclusion. □

One function, the identity function, is particularly important.

Definition 100.11 (Identity Function)**.** *The identity function* id_A *from a set* A *to itself, is the set* $\{(x,x) : x \in A\}$.

Lemma 100.12. *For any function* f *from* A *to* B *we have*

$$f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f.$$

Proof. Let x be an element of A, then

$$(f \circ \mathrm{id}_A)(x) = f(\mathrm{id}_A(x)) = f(x) = \mathrm{id}_B(f(x)) = (\mathrm{id}_B \circ f)(x).$$

□

Injective, Surjective and Bijective Functions: Two properties that we shall use all the time are the following.

Definition 100.13 (Injective and Surjective)**.**
A function f *from* A *to* B *is said to be injective (or 1:1) if for any two elements,* x, y *of* A *with* $f(x) = f(y)$ *we have* $x = y$.
A function f *from* A *to* B *is said to be surjective (or onto) if for every element* $b \in B$ *there exists an element* $a \in A$ *such that* $f(a) = b$.

A function which is both injective and surjective is called bijective (or a 1:1 correspondence). We shall now give some examples.

(1) The function from $\mathbb{R}$ to $\mathbb{R}$ given by $f(x) = x + 2$ is bijective.
(2) The function from $\mathbb{N}$ to $\mathbb{N}$ given by $f(x) = x + 2$ is injective but not surjective as the elements $\{0, 1\}$ are not the image of anything.
(3) The function from $\mathbb{R}$ to $\mathbb{R}_{\geq 0}$ given by $f(x) = x^2$ is surjective as every non-negative real number has a square root in $\mathbb{R}$ but it is not injective as if $x^2 = y^2$ then we could have $x = -y$.

The following gives us a good reason to study bijective functions.

Lemma 100.14. *A function* $f : A \to B$ *is bijective if and only if there exists a function* $g : B \to A$ *such that* $f \circ g$ *and* $g \circ f$ *are the identity function.*

We leave the proof of this lemma as an exercise. Note that applying this lemma to the resulting g means that g is also bijective. Such a function as g in the above lemma is called the inverse of f and is usually denoted f^{-1}. Note that a function only has an inverse if it is bijective.

A.4. Permutations

We let A be a finite set of cardinality n; without loss of generality we can assume that $A = \{1, 2, \ldots, n\}$. A bijective function from A to A is called a permutation. The set of all permutations on a set of cardinality n is denoted by S_n.

Suppose $A = \{1, 2, 3\}$, then we have the permutation $f(1) = 2$, $f(2) = 3$ and $f(3) = 1$. This is a very cumbersome way to write a permutation. Mathematicians (being lazy people) have invented the following notation: the function f above is written as

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}.$$

What should be noted about this notation (which applies for arbitrary n) is that all the numbers between 1 and n occur exactly once on each row. The first row is always given as the numbers 1 to n in increasing order. Any such matrix with these properties represents a permutation, and all permutations can be represented by such a matrix. This leads us to the following elementary result.

Lemma 100.15. *The cardinality of the set S_n is $n!$.*

PROOF. This is a well-known argument. There are n choices for the first element in the second row of the above matrix. Then there are $n-1$ choices for the second element in the second row and so on. □

If σ is a permutation on a set S then we usually think of σ acting on the set. So if $s \in S$ then we write s^σ or $\sigma(s)$ for the action of σ on the element s.

Suppose we define the permutations

$$\begin{aligned} g &= \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}, \\ f &= \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}. \end{aligned}$$

As permutations are nothing but functions we can compose them. Remembering that $g \circ f$ means apply the function f and then apply the function g we see that

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

means $1 \to 3 \to 1$, $2 \to 2 \to 3$ and $3 \to 1 \to 2$. Hence, the result of composing the above two permutations is

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}. \tag{24}$$

However, this can cause confusion when using our "acting on a set" notation above. For example

$$1^{g \circ f} = g(f(1)) = 3$$

so we are unable to read the permutation from left to right. However, if we use another notation, say $\cdot$, to mean

$$f \cdot g = g \circ f$$

then we are able to read the expression from left to right. We shall call this operation multiplying permutations.

Cycle Notation: Mathematicians, as we said, are by nature lazy people and this notation we have introduced is still a little too much. For instance we always write down the numbers $1, \ldots, n$ in the top row of each matrix to represent a permutation. Also some columns are redundant, for instance the first column of the permutation in equation (24). We now introduce another notation for permutations which is concise and clear. We first need to define what a cycle is.

Definition 100.16 (Cycle)**.** *By a cycle or n-cycle we mean the object $(x_1, \ldots, x_n)$ with distinct $x_i \in \mathbb{N} \setminus \{0\}$. This represents the permutation $f(x_1) = x_2$, $f(x_2) = x_3$, $\ldots$, $f(x_{n-1}) = x_n$, $f(x_n) = x_1$ and for $x \notin \{x_1, \ldots, x_n\}$ we have $f(x) = x$.*

For instance we have

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = (1,2,3) = (2,3,1) = (3,1,2).$$

Notice that a cycle is not a unique way of representing a permutation. Most permutations cannot be written as a single cycle, but they can be written as a product of cycles. For example we have

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = (1,3) \circ (2) = (3,1) \circ (2).$$

The identity permutation is represented by (). Again, as mathematicians are lazy we always write $(1,3) \circ (2) = (1,3)$. This can lead to ambiguities as $(1,2)$ could represent a function from

$$\{1,2\} \text{ to } \{1,2\}$$

or

$$\{1,2,\ldots,n\} \text{ to } \{1,2,\ldots,n\}.$$

However, which function it represents is usually clear from the context.

Two cycles $(x_1,\ldots,x_n)$ and $(y_1,\ldots,y_n)$ are called disjoint if $\{x_1,\ldots,x_n\} \cap \{y_1,\ldots,y_n\} = \emptyset$. It is easy to show that if σ and τ are two disjoint cycles then

$$\sigma \cdot \tau = \tau \cdot \sigma.$$

Note that this is not true for cycles which are not disjoint, e.g.

$$(1,2,3,4) \cdot (3,5) = (1,2,5,3,4) \neq (1,2,3,5,4) = (3,5) \cdot (1,2,3,4).$$

Our action of permutations on the underlying set can now be read easily from left to right,

$$2^{(1,2,3,4)\cdot(3,5)} = 3^{(3,5)} = 5 = 2^{(1,2,5,3,4)},$$

as the permutation $(1,2,3,4)$ maps 2 to 3 and the permutation $(3,5)$ maps 3 to 5.

What really makes disjoint cycles interesting is the following.

Lemma 100.17. *Every permutation can be written as a product of disjoint cycles.*

Proof. Let σ be a permutation on $\{1,\ldots,n\}$. Let σ_1 denote the cycle

$$(1, \sigma(1), \sigma(\sigma(1)), \ldots, \sigma(\ldots\sigma(1)\ldots)),$$

where we keep applying σ until we get back to 1. We then take an element x of $\{1,\ldots,n\}$ such that $\sigma_1(x) = x$, if one exists, and consider the cycle σ_2 given by

$$(x, \sigma(x), \sigma(\sigma(x)), \ldots, \sigma(\ldots\sigma(x)\ldots)).$$

We then take an element of $\{1,\ldots,n\}$ which is fixed by σ_1 and σ_2 to create a cycle σ_3. We continue this way until we have used all elements of $\{1,\ldots,n\}$. The resulting cycles $\sigma_1,\ldots,\sigma_t$ are obviously disjoint and their product is equal to the cycle σ. □

What is nice about this proof is that it is constructive. Given a permutation we can follow the procedure in the proof to obtain the permutation as a product of disjoint cycles. Consider the permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 3 & 7 & 6 & 8 & 4 & 1 & 5 & 9 \end{pmatrix}.$$

We have $\sigma(1) = 2,\ \sigma(2) = 3,\ \sigma(3) = 7$ and $\sigma(7) = 1$ so the first cycle is

$$\sigma_1 = (1,2,3,7).$$

The next element of $\{1,\ldots,9\}$ which we have not yet considered is 4. We have $\sigma(4) = 6$ and $\sigma(6) = 4$ so $\sigma_2 = (4,6)$. Continuing in this way we find $\sigma_3 = (5,8)$ and $\sigma_4 = (9)$. Hence we have

$$\sigma = (1,2,3,7)(4,6)(5,8)(9) = (1,2,3,7)(4,6)(5,8).$$

A.5. Operations

In mathematics one meets lots of binary operations: ordinary addition and multiplication, composition of functions, matrix addition and multiplication, multiplication of permutations, etc., the list is somewhat endless. All of these binary operations have a lot in common; they also have many differences, for instance, for two real numbers x and y we have $x \cdot y = y \cdot x$, but for two 2×2 matrices with real entries, A and B, it is not true that we always have $A \cdot B = B \cdot A$. To study the similarities and differences between these operations we formalize the concept below. We then prove some results which are true of operations given some basic properties, these results can then be applied to any of the operations above which satisfy the given properties. Hence our abstraction will allow us to prove results in many areas at once.

Definition 100.18 (Operation)**.** *A (binary) operation on a set A is a function from the domain $A \times A$ to the codomain A.*

So if $A = \mathbb{R}$ we could have the function $f(x, y) = x + y$. Writing $f(x, y)$ all the time can become a pain so we often write a symbol between the x and the y to denote the operation, e.g.

$$\begin{array}{ccc} x \cdot y & x + y & x \oplus y \\ x \circ y & x \odot y & x \diamond y \\ x \wedge y & x \vee y & x \star y. \end{array}$$

Most often we write $x + y$ and $x \cdot y$; we refer to the former as additive notation and the latter as multiplicative notation. One should bear in mind that we may not be actually referring to ordinary multiplication and addition when we use these terms/notations.

Associative and Commutative: Operations can satisfy various properties.

Definition 100.19 (Associative)**.** *An operation $\diamond$ is said to be associative if for all x, y and z we have*

$$(x \diamond y) \diamond z = x \diamond (y \diamond z).$$

Operations which are associative include all the examples mentioned above. Non-associative operations do exist (for example the subtraction operation on the integers is non-associative) but we shall not be interested in them much. Note that for an associative operation the expression

$$w \diamond x \diamond y \diamond z$$

is well defined; as long as we do not change the relative position of any of the terms it does not matter which operation we carry out first.

Definition 100.20 (Commutative)**.** *An operation $\vee$ is said to be commutative if for all x and y we have*

$$x \vee y = y \vee x.$$

Ordinary addition, multiplication and matrix addition are commutative, but multiplication of matrices and permutations are not.

Identities:

Definition 100.21 (Identity)**.** *An operation $\cdot$ on the set A is said to have an identity if there exists an element e of A such that for all x we have*

$$e \cdot x = x \cdot e = x.$$

The first thing we notice is that all the example operations above possess an identity, but ordinary subtraction on the set $\mathbb{R}$ does not possess an identity. The following shows that there can be at most one identity for any given operation.

Lemma 100.22. *If an identity exists then it is unique. It is then called "the" identity.*

PROOF. Suppose there are two identities e and e'. As e is an identity we have $e \cdot e' = e'$ and as e' is an identity we have $e \cdot e' = e$. Hence, we have $e' = e \cdot e' = e$. □

Usually if we are using an additive notation then we denote the identity by 0 to correspond with the identity for ordinary addition, and if we are using the multiplicative notation then we denote the identity by either 1 or e.

Inverses:

Definition 100.23 (Inverses)**.** *Let $+$ be an operation on a set A with identity 0. Let $x \in A$. If there is a $y \in A$ such that*

$$x + y = y + x = 0$$

then we call y an inverse of x.

In the additive notation it is usual to write the inverse of x as $-x$. In the multiplicative notation it is usual to write the inverse as x^{-1}.

All elements in $\mathbb{R}$ have inverses with respect to ordinary addition. All elements in $\mathbb{R}$ except zero have inverses with respect to ordinary multiplication. Every permutation has an inverse with respect to multiplication of permutations. However, only square matrices of non-zero determinant have inverses with respect to matrix multiplication. The next result shows that an element can have at most one inverse assuming the operation is associative.

Lemma 100.24. *Consider an associative operation on a set A with identity e. Let $x \in A$ have an inverse y, then this inverse is unique, we call it "the" inverse.*

PROOF. Suppose there are two such inverses y and y', then

$$y = y \cdot e = y \cdot (x \cdot y') = (y \cdot x) \cdot y' = e \cdot y' = y'.$$

Note how we used the associativity property above. □

Lemma 100.25. *Consider an associative operation on a set A with an identity e. If $a, b, x \in A$ with $a \cdot x = b \cdot x$ then $a = b$.*

PROOF. Let y denote the inverse of x, then we have

$$a = a \cdot e = a \cdot (x \cdot y) = (a \cdot x) \cdot y = (b \cdot x) \cdot y = b \cdot (x \cdot y) = b \cdot e = b.$$

□

We shall assume from now on that all operations we shall encounter are associative.

Powers: Say one wishes to perform the same operation over and over again, for example

$$x \vee x \vee x \vee \cdots \vee x \vee x.$$

If our operation is written additively then we write for $n \in \mathbb{N}$, $n \cdot x$ for $x + \cdots + x$, whilst if our operation is written multiplicatively we write x^n for $x \cdots x$. The following result can then be proved by induction.

Lemma 100.26 (Law of Powers)**.** *For any operation $\circ$ which is associative we have*

$$g^m \circ g^n = g^{m+n}, \ (g^m)^n = g^{m \cdot n}.$$

We can extend the notation to all $n \in \mathbb{Z}$ if x has an inverse (and the operation an identity), by $(-n) \cdot x = n \cdot (-x)$ and $x^{-n} = (x^{-1})^n$. The following lemma is obvious, but often causes problems as it is slightly counter-intuitive. To get it in your brain consider the case of matrices.

Lemma 100.27. *Consider a set with an associative operation which has an identity, e. If $x, y \in G$ possess inverses then we have*

(1) $(x^{-1})^{-1} = x$.
(2) $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$.

PROOF. For the first we notice

$$x^{-1} \cdot x = e = x \cdot x^{-1}.$$

Hence by definition of inverses the result follows. For the second we have

$$x \cdot y \cdot (y^{-1} \cdot x^{-1}) = x \cdot (y \cdot y^{-1}) \cdot x^{-1} = x \cdot e \cdot x^{-1} = x \cdot x^{-1} = e,$$

and again the result follows by the definition of inverses. □

We have the following dictionary to translate between additive and multiplicative notations:

Additive	Multiplicative
$x + y$	$x \cdot y$
0	1 or e
$-x$	x^{-1}
$n \cdot x$	x^n

A.6. Groups

Definition 100.28 (Group)**.** *A group is a set G with a binary operation $\circ$ such that*

(1) $\circ$ *is associative.*
(2) $\circ$ *has an identity element in G.*
(3) *Every element of G has an inverse.*

Note that we have not said that the binary operation is closed as this is implicit in our definition of what an operation is. If the operation is also commutative then we say that we have a commutative, or abelian, group. The following are all groups; as an exercise you should decide on the identity element, what the inverse of each element is, and which groups are abelian.

(1) The integers $\mathbb{Z}$ under addition (written $\mathbb{Z}^+$).
(2) The rationals $\mathbb{Q}$ under addition (written $\mathbb{Q}^+$).
(3) The reals $\mathbb{R}$ under addition (written $\mathbb{R}^+$).
(4) The complex numbers $\mathbb{C}$ under addition (written $\mathbb{C}^+$).
(5) The rationals (excluding zero) $\mathbb{Q} \setminus \{0\}$ under multiplication (written $\mathbb{Q}^*$).
(6) The reals (excluding zero) $\mathbb{R} \setminus \{0\}$ under multiplication (written $\mathbb{R}^*$).
(7) The complex numbers (excluding zero) $\mathbb{C} \setminus \{0\}$ under multiplication (written $\mathbb{C}^*$).
(8) The set of n-ary vectors over $\mathbb{Z}, \mathbb{Q}, \ldots$, etc. under vector addition.
(9) The set of $n \times m$ matrices with integer, rational, real or complex entries under matrix addition. This set is written $M_{n\times m}(\mathbb{Z})$, etc. however when $m = n$ we write $M_n(\mathbb{Z})$ instead of $M_{n\times n}(\mathbb{Z})$.
(10) The general linear group (the matrices of non-zero determinant) over the rationals, reals or complex numbers under matrix multiplication (written $\mathrm{GL}_n(\mathbb{Q})$, etc.).
(11) The special linear group (the matrices of determinant ± 1) over the integers, rationals etc. (written $\mathrm{SL}_n(\mathbb{Z})$, etc.).
(12) The set of permutations on n elements, written S_n and often called the symmetric group on n letters.
(13) The set of continuous (differentiable) functions from $\mathbb{R}$ to $\mathbb{R}$ under pointwise addition.

The list is endless; a group is one of the most basic concepts in mathematics. However, not all mathematical objects are groups. Consider the following list of sets and operations which are not groups, you should also decide why.

(1) The natural numbers $\mathbb{N}$ under ordinary addition or multiplication.
(2) The integers $\mathbb{Z}$ under subtraction or multiplication.

We now give a number of definitions related to groups.

Definition 100.29 (Orders)**.**

The order of a group is the number of elements in the underlying set G and is denoted $|G|$ or $\#G$. The order of a group can be infinite.
The order of an element $g \in G$ is the least positive integer n such that $g^n = e$, if such an n exists; otherwise we say that g has infinite order.

Definition 100.30 (Cyclic Groups and Generators)**.**
A cyclic group G is a group which has an element g such that each element of G can be written in the form g^n for some $n \in \mathbb{Z}$ (in multiplicative notation). If this is the case then one can write $G = \langle g \rangle$ and one says that g is a generator of the group G.

Note that the only element in a group with order one is the identity element and if x is an element of a group then x and x^{-1} have the same order.

Lemma 100.31. *If $G = \langle g \rangle$ and g has finite order n then the order of G is n.*

Proof. Every element of G can be written as g^m for some $m \in \mathbb{Z}$, but as g has order n there are only n distinct such values, as

$$g^{n+1} = g^n \circ g = e \circ g = g.$$

So the group G has only n elements. □

Let us relate this back to the permutations which we introduced earlier. Recall that the set of permutations on a fixed set S forms a group under composition. It is easy to see that if $\sigma \in S_n$ is a k-cycle then σ has order k in S_n. One can also easily see that if σ is a product of disjoint cycles then the order of σ is the least common multiple of the orders of the constituent cycles.

A subset S of G is said to generate G if every element of G can be written as a product of elements of S. For instance

- the group S_3 is generated by the set $\{(1,2),(1,2,3)\}$,
- the group $\mathbb{Z}^+$ is generated by the element 1,
- the group $\mathbb{Q}^*$ is generated by the set of prime numbers, it therefore has an infinite number of generators.

Note that the order of a group says nothing about the number of generators it has, although the order is clearly a trivial upper bound on the number of generators.

An important set of finite groups which are easy to understand is groups obtained by considering the integers modulo a number m. Recall that we have $\mathbb{Z}/m\mathbb{Z} = \{0, 1, \ldots, m-1\}$. This is a group with respect to addition, when we take the non-negative remainder after forming the sum of two elements. It is not a group with respect to multiplication in general, even when we exclude 0. We can, however, get around this by setting

$$(\mathbb{Z}/m\mathbb{Z})^* = \{x \in \mathbb{Z}/m\mathbb{Z} : \gcd(m, x) = 1\}.$$

This latter set is a group with respect to multiplication, when we take the non-negative remainder after forming the product of two elements. The order of $(\mathbb{Z}/m\mathbb{Z})^*$ is denoted $\phi(m)$, the Euler ϕ function. This is an important function in the theory of numbers. As an example we have

$$\phi(p) = p - 1,$$

if p is a prime number. We shall return to this function later.

Subgroups: We now turn our attention to subgroups.

Definition 100.32 (Subgroup)**.** *A* subgroup *H of a group G is a subset of G which is also a group with respect to the operation of G. We write in this case $H < G$. A subgroup H is called* trivial *if it is equal to the whole group G, or is equal to the group consisting of just the identity element.*

Note that by this definition $\mathrm{GL}_n(\mathbb{R})$ is not a subgroup of $M_n(\mathbb{R})$, although $\mathrm{GL}_n(\mathbb{R}) \subset M_n(\mathbb{R})$. The operation on $\mathrm{GL}_n(\mathbb{R})$ is matrix multiplication whilst that on $M_n(\mathbb{R})$ is matrix addition. However we do have the subgroup chains:

$$\mathbb{Z}^+ < \mathbb{Q}^+ < \mathbb{R}^+ < \mathbb{C}^+,$$
$$\mathbb{Q}^* < \mathbb{R}^* < \mathbb{C}^*.$$

If we also identify $x \in \mathbb{Z}$ with the diagonal matrix $\mathrm{diag}(x, \ldots, x)$ then we also have that $\mathbb{Z}^+$ is a subgroup of $M_n(\mathbb{Z})$ and so on.

As an important example, consider the set $2\mathbb{Z}$ of even integers, which is a subgroup of $\mathbb{Z}^+$. If we write $\mathbb{Z}^+ = 1\mathbb{Z}$, then we have $n\mathbb{Z} < m\mathbb{Z}$ if and only if m divides n, where

$$m\mathbb{Z} = \{\ldots, -2 \cdot m, -m, 0, m, 2 \cdot m, \ldots\}.$$

We hence obtain various chains of subgroups of $\mathbb{Z}^+$,

$$18\mathbb{Z} < 6\mathbb{Z} < 2\mathbb{Z} < \mathbb{Z}^+,$$
$$18\mathbb{Z} < 9\mathbb{Z} < 3\mathbb{Z} < \mathbb{Z}^+,$$
$$18\mathbb{Z} < 6\mathbb{Z} < 3\mathbb{Z} < \mathbb{Z}^+.$$

We now show that these are the only such subgroups of $\mathbb{Z}^+$.

Lemma 100.33. *The only subgroups of $\mathbb{Z}^+$ are $n\mathbb{Z}$ for some positive integer n.*

Proof. Let H be a subgroup of $\mathbb{Z}^+$. As H is non-empty it must contain an element x and its inverse $-x$. Hence H contains at least one positive element n. Let n denote the least such positive element of H. Hence $n\mathbb{Z} \subseteq H$.

Now let m denote an arbitrary non-zero element of H. By Euclidean division, there exist $q, r \in \mathbb{Z}$ with $0 \leq r < n$ such that

$$m = q \cdot n + r.$$

Hence $r \in H$. By choice of n this must mean $r = 0$, since H is a group under addition. Therefore all elements of H are of the form $n \cdot q$, for some value of q, which is what was required. □

So every subgroup of $\mathbb{Z}^+$ is an infinite cyclic group. This last lemma combined with the earlier subgroup chains gives us a good definition of what a prime number is.

Definition 100.34 (Prime Number)**.** *A prime number is a (positive) generator of a non-trivial subgroup H of $\mathbb{Z}^+$, for which no subgroup of $\mathbb{Z}^+$ contains H except $\mathbb{Z}^+$ and H itself.*

What is good about this definition is that we have not referred to the multiplicative structure of $\mathbb{Z}$ to define the primes. Also it is obvious that neither zero nor one is a prime number. You should convince yourself that this definition leads to the usual definition of primes in terms of divisibility. In addition the above definition allows one to generalize the notion of primality to other settings; for how this is done consult any standard textbook on abstract algebra.

Normal Subgroups and Cosets: A normal subgroup is particularly important in the theory of groups. The name should not be thought of as meaning that these are the subgroups that normally arise; the name is a historic accident. To define a normal subgroup we first need to define what is meant by conjugate elements.

Definition 100.35 (Conjugate)**.** *Two elements x, y of a group G are said to be conjugate if there is an element $g \in G$ such that $x = g^{-1} \cdot y \cdot g$.*

It is obvious that two conjugate elements have the same order. As an exercise you should show that the conjugates in a group form an equivalence class under the conjugate relation. If N is a subgroup of G we define, for any $g \in G$,

$$g^{-1}Ng = \{g^{-1} \cdot x \cdot g : x \in N\},$$

which is another subgroup of G, called a conjugate of the subgroup N.

Definition 100.36 (Normal Subgroup)**.** *A subgroup $N < G$ is said to be a normal subgroup if $g^{-1}Ng \subseteq N$ for all $g \in G$. If this is the case then we write $N \lhd G$.*

For any group G we have $G \lhd G$ and $\{e\} \lhd G$ and if G is an abelian group then every subgroup of G is normal. The importance of normal subgroups comes from the fact that these are subgroups by which we can factor out. This is related to the cosets of a subgroup which we now go on to introduce.

Definition 100.37 (Cosets)**.** *Let G be a group and $H < G$ (H is not necessarily normal). Fix an element $g \in G$, then we define the left coset of H with respect to g to be the set*

$$gH = \{g \cdot h : h \in H\}.$$

Similarly we define the right coset of H with respect to g to be the set

$$Hg = \{h \cdot g : h \in H\}.$$

Let H denote a subgroup of G then one can show that the set of all left (or right) cosets of H in G forms a partition of G, but we leave this to the reader. In addition if $a, b \in G$ then $aH = bH$ if and only if $a \in bH$, which is also equivalent to $b \in aH$, a fact which we also leave to the reader to show. Note that we can have two equal cosets $aH = bH$ without having $a = b$.

What these latter facts show is that if we define the relation R_H on the group G with respect to the subgroup H by

$$(a, b) \in R_H \text{ if and only if } a = b \cdot h \text{ for some } h \in H,$$

then this relation is an equivalence relation. The equivalence classes are just the left cosets of H in G.

The number of left cosets of a subgroup H in G is denoted by $(G : H)_L$, the number of right cosets is denoted by $(G : H)_R$. We are now in a position to prove the most important theorem of elementary group theory, namely Lagrange's Theorem.

Theorem 100.38 (Lagrange's Theorem)**.** *Let H be a subgroup of a finite group G then*

$$\begin{aligned} |G| &= (G : H)_L \cdot |H| \\ &= (G : H)_R \cdot |H|. \end{aligned}$$

Before we prove this result we state some obvious important corollaries.

Corollary 100.39.

- *We have $(G : H)_L = (G : H)_R$; we denote this common number by $(G : H)$ and call it the index of the subgroup H in G.*
- *The order of a subgroup and the index of a subgroup both divide the order of the group.*
- *If G is a group of prime order, then G has only the subgroups G and $\langle e \rangle$.*

We now return to the proof of Lagrange's Theorem.

Proof. We form the following collection of distinct left cosets of H in G which we define inductively. Put $g_1 = e$ and assume we are given i cosets by $g_1H, \ldots, g_iH$. Now take an element g_{i+1} not lying in any of the left cosets g_jH for $j \leq i$. After a finite number of such steps we have exhausted the elements of the group G. So we have a disjoint union of left cosets which cover the whole group.

$$G = \bigcup_{1 \leq i \leq (G:H)_L} g_iH.$$

We also have for each i, j that $|g_iH| = |g_jH|$, this follows from the fact that the map

$$\begin{aligned} H &\longrightarrow gH \\ h &\longmapsto g \cdot h \end{aligned}$$

is a bijective map on sets. Hence

$$|G| = \sum_{1 \le i \le (G:H)_L} |g_i H| = (G:H)_L |H|.$$

The other equality follows using the same argument. □

We can also deduce from the corollaries the following.

Lemma 100.40. *If G is a group of prime order then it is cyclic.*

Proof. If $g \in G$ is not the identity then $\langle g \rangle$ is a subgroup of G of order ≥ 2. But then it must have order $|G|$ and so G is cyclic. □

We can use Lagrange's Theorem to write down the subgroups of some small groups. For example, consider the group S_3: this has order 6 so by Lagrange's Theorem its subgroups must have order $1, 2, 3$ or 6. It is easy to see that the only subgroups are therefore:

- One subgroup of order 1; namely $\langle (1) \rangle$,
- Three subgroups of order 2; namely $\langle (1,2) \rangle$, $\langle (1,3) \rangle$ and $\langle (2,3) \rangle$,
- One subgroup of order 3; namely $\langle (1,2,3) \rangle$,
- One subgroup of order 6, which is S_3 obviously.

Factor or Quotient Groups: We let G be a group with a normal subgroup N. The following elementary lemma, whose proof we again leave to the reader, gives us our justification for looking at normal subgroups.

Lemma 100.41. *Let $H < G$ then the following are equivalent:*

1. *$xH = Hx$ for all $x \in G$.*
2. *$x^{-1}Hx = H$ for all $x \in G$.*
3. *$H \lhd G$.*
4. *$x^{-1} \cdot h \cdot x \in H$ for all $x \in G$ and $h \in H$.*

By G/N we denote the set of left cosets of N; note that these are the same as the right cosets of N. We note that two cosets, $g_1 N$ and $g_2 N$ are equal if and only if $g_1^{-1} g_2 \in N$.

We wish to turn G/N into a group, the so-called factor group or quotient group. Let $g_1 N$ and $g_2 N$ denote any two elements of G/N, then we define the product of their left cosets to be $(g_1 g_2)N$.

We first need to show that this is a well-defined operation, i.e. if we replace g_1 by g_1' and g_2 by g_2' with $g_1^{-1} g_1' = n_1 \in N$ and $g_2^{-1} \cdot g_2' = n_2 \in N$ then our product still gives the same coset. In other words we wish to show

$$(g_1 \cdot g_2)N = (g_1' \cdot g_2')N.$$

Now let $x \in (g_1 \cdot g_2)N$, then $x = g_1 \cdot g_2 \cdot n$ for some $n \in N$. Then $x = g_1' \cdot n_1^{-1} \cdot g_2' \cdot n_2^{-1} \cdot n$. But as G is normal (left cosets = right cosets) we have ${n_1'}^{-1} \cdot g_2' = g_2' \cdot n_3$ for some $n_3 \in N$. Hence

$$x = g_1' \cdot g_2' \cdot n_3 \cdot n_2^{-1} \cdot n \in (g_1' \cdot g_2')N.$$

This proves the first inclusion; the other follows similarly. We conclude that our operation on G/N is well defined. One can also show that if N is an arbitrary subgroup of G and we define the operation on the cosets above then this is only a well-defined operation if N is a normal subgroup of G.

So we have a well-defined operation on G/N, we now need to show that this operation satisfies the axioms of a group:

- As an identity we take $eN = N$, since for all $g \in G$ we have
$$eN \cdot gN = (e \cdot g)N = gN.$$
- As an inverse of (gN) we take $g^{-1}N$ as
$$gN \cdot g^{-1}N = (g \cdot g^{-1})N = eN = N.$$

- Associativity follows from

$$\begin{aligned}(g_1N)\cdot(g_2N\cdot g_3N) &= g_1N\cdot((g_2\cdot g_3)N) = (g_1\cdot(g_2\cdot g_3))N\\ &= ((g_1\cdot g_2)\cdot g_3)N = ((g_1\cdot g_2)N)\cdot g_3N\\ &= (g_1N\cdot g_2N)\cdot(g_3N).\end{aligned}$$

We now present some examples.

(1) Let G be an arbitrary finite group of order greater than one; let H be a subgroup of G. Then $H = G$ and $H = \{e\}$ are always normal subgroups of G.
(2) If $H = G$ then there is only one coset and so we have $G/G = \{G\}$ is a group of order one.
(3) If $H = \{e\}$ then the cosets of H are the one-element subsets of G. That is $G/\{e\} = \{\{g\} : g \in G\}$.
(4) Put $G = S_3$ and $N = \{(1), (1,2,3), (1,3,2)\}$, then N is a normal subgroup of G. The cosets of N in G are N and $(1,2)N$ with
$$((1,2)N)^2 = (1,2)^2N = (1)N = N.$$
Hence $S_3/\langle(1,2,3)\rangle$ is a cyclic group of order 2.
(5) If G is abelian then every subgroup H of G is normal, so one can always form the quotient group G/H.
(6) Since $(\mathbb{Z}, +)$ is abelian we have that $m\mathbb{Z}$ is always a normal subgroup. Forming the quotient group $\mathbb{Z}/m\mathbb{Z}$ we obtain the group of integers modulo m under addition.

Homomorphisms: Let G_1 and G_2 be two groups; we wish to look at the functions from G_1 to G_2. Obviously we could look at all such functions, however by doing this we would lose all the structure that the group laws give us. We restrict ourselves to maps which preserve these group laws.

Definition 100.42 (Homomorphism)**.** *A homomorphism from a group G_1 to a group G_2 is a function f with domain G_1 and codomain G_2 such that for all $x, y \in G_1$ we have*

$$f(x\cdot y) = f(x)\cdot f(y).$$

Note that multiplication on the left is with the operation of the group G_1 whilst the multiplication on the right is with respect to the operation of G_2. As examples we have

(1) The identity map $\mathrm{id}_G : G \to G$, where $\mathrm{id}_G(g) = g$ is a group homomorphism.
(2) Consider the function $\mathbb{R}^+ \to \mathbb{R}^*$ given by $f(x) = e^x$. This is a homomorphism as for all $x, y \in \mathbb{R}$ we have
$$e^{x+y} = e^x \cdot e^y.$$
(3) Consider the map from $\mathbb{C}^*$ to $\mathbb{R}^*$ given by $f(z) = |z|$. This is also a homomorphism.
(4) Consider the map from $\mathrm{GL}_n(\mathbb{C})$ to $\mathbb{C}^*$ given by $f(A) = \det(A)$; this is a group homomorphism as $\det(A\cdot B) = \det(A)\cdot\det(B)$ for any two elements of $\mathrm{GL}_n(\mathbb{C})$.

Two elementary properties of homomorphisms are summarized in the following lemma.

Lemma 100.43. *Let $f : G_1 \to G_2$ be a homomorphism of groups, then*

(1) $f(e_1) = e_2$.
(2) *For all $x \in G_1$ we have $f(x^{-1}) = (f(x))^{-1}$.*

Proof. For the first result we have $e_2 \cdot f(x) = f(x) = f(e_1 \cdot x) = f(e_1)\cdot f(x)$, and then from Lemma 100.25 we have $e_2 = f(e_1)$ as required. For the second result we have

$$f(x^{-1})\cdot f(x) = f(x^{-1}\cdot x) = f(e_1) = e_2,$$

so the result follows by definition. □

For any homomorphism f from G_1 to G_2 there are two special subgroups associated with f.
Definition 100.44 (Kernel and Image)**.**

- *The kernel of f is the set*
$$\mathrm{Ker}f = \{x \in G_1 : f(x) = e_2\}.$$
- *The image of f is the set*
$$\mathrm{Im}f = \{y \in G_2 : y = f(x),\ x \in G_1\}.$$

Lemma 100.45. $\mathrm{Ker}f$ *is a normal subgroup of* G_1.

Proof. We first show that it is a subgroup. It is certainly non-empty as $e_1 \in \mathrm{Ker}f$ as $f(e_1) = e_2$. Now if $x \in \mathrm{Ker}f$ then $f(x^{-1}) = f(x)^{-1} = e_2^{-1} = e_2$, hence $x^{-1} \in \mathrm{Ker}f$. Hence to show that $\mathrm{Ker}f$ is a subgroup we only have to show that for all $x, y \in \mathrm{Ker}f$ we have $x \cdot y^{-1} \in \mathrm{Ker}f$. But this is easy as if $x, y \in \mathrm{Ker}f$ then we have
$$f(x \cdot y^{-1}) = f(x) \cdot f(y^{-1}) = e_2 \cdot e_2 = e_2,$$
and we are done.

We now show that $\mathrm{Ker}f$ is in fact a normal subgroup of G_1. We need to show that if $x \in \mathrm{Ker}f$ then $g^{-1} \cdot x \cdot g \in \mathrm{Ker}f$ for all $g \in G_1$. So let $x \in \mathrm{Ker}f$ and let $g \in G_1$, then we have
$$f(g^{-1} \cdot x \cdot g) = f(g^{-1}) \cdot f(x) \cdot f(g) = f(g)^{-1} \cdot e_2 \cdot f(g) = f(g)^{-1} \cdot f(g) = e_2,$$
so we are done. □

Lemma 100.46. $\mathrm{Im}f$ *is a subgroup of* G_2.

Proof. $\mathrm{Im}f$ is certainly non-empty as $f(e_1) = e_2$. Now suppose $y \in \mathrm{Im}f$ so there is an $x \in G_2$ such that $f(x) = y$, then $y^{-1} = f(x)^{-1} = f(x^{-1})$ and $x^{-1} \in G_1$ so $y^{-1} \in \mathrm{Im}f$. Now suppose $y_1, y_2 \in \mathrm{Im}f$, hence for some x_1, x_2 we have
$$y_1 \cdot y_2^{-1} = f(x_1) \cdot f(x_2^{-1}) = f(x_1 \cdot x_2^{-1}).$$
Hence $\mathrm{Im}f < G_2$. □

It is clear that $\mathrm{Im}f$ in some sense measures whether the homomorphism f is surjective as f is surjective if and only if $\mathrm{Im}f = G_2$. Actually the set $G_2/\mathrm{Im}f$ is a better measure of the surjectivity of the function. On the other hand, $\mathrm{Ker}f$ measures how far from injective f is, due to the following result.

Lemma 100.47. *A homomorphism, f, is injective if and only if* $\mathrm{Ker}f = \{e_1\}$.

Proof. Assume f is injective, then we know that if $f(x) = e_2 = f(e_1)$ then $x = e_1$ and so $\mathrm{Ker}f = \{e_1\}$. Now assume that $\mathrm{Ker}f = \{e_1\}$ and let $x, y \in G_1$ be such that $f(x) = f(y)$. Then
$$f(x \cdot y^{-1}) = f(x) \cdot f(y^{-1}) = f(x) \cdot f(y)^{-1} = f(y) \cdot f(y)^{-1} = e_2.$$
So $x \cdot y^{-1} \in \mathrm{Ker}f$, but then $x \cdot y^{-1} = e_1$ and so $x = y$. So f is injective. □

Isomorphisms: Bijective homomorphisms allow us to categorize groups more effectively, as the following definition elaborates.

Definition 100.48 (Isomorphism)**.** *A homomorphism f is said to be an isomorphism if it is bijective. Two groups are said to be isomorphic if there is an isomorphism between them, in which case we write $G_1 \cong G_2$.*

Note that this means that isomorphic groups have the same number of elements. Indeed for all intents and purposes one may as well assume that isomorphic groups are equal, since they look the same up to relabelling of elements. Isomorphisms satisfy the following properties.

- If $f : G_1 \to G_2$ and $g : G_2 \to G_3$ are isomorphisms then $g \circ f$ is also an isomorphism, i.e. isomorphisms are transitive.
- If $f : G_1 \to G_2$ is an isomorphism then so is $f^{-1} : G_2 \to G_1$, i.e. isomorphisms are symmetric.

- The identity map $\mathsf{id} : G \to G$ given by $\mathsf{id}(x) = x$ is an isomorphism, i.e. isomorphisms are reflexive.

From this we see that the relation "is isomorphic to" is an equivalence relation on the class of all groups. This justifies our notion of isomorphic being like equal.

Let G_1, G_2 be two groups, then we define the product group $G_1 \times G_2$ to be the set $G_1 \times G_2$ of ordered pairs (g_1, g_2) with $g_1 \in G_1$ and $g_2 \in G_2$. The group operation on $G_1 \times G_2$ is given componentwise:

$$(g_1, g_2) \circ (g_1', g_2') = (g_1 \circ g_1', g_2 \circ g_2').$$

The first $\circ$ refers to the group $G_1 \times G_2$, the second to the group G_1 and the third to the group G_2. Some well-known groups can actually be represented as product groups. For example, consider the map

$$\begin{aligned} \mathbb{C}^+ &\longrightarrow \mathbb{R}^+ \times \mathbb{R}^+ \\ z &\longmapsto (\mathrm{Re}(z), \mathrm{Im}(z)). \end{aligned}$$

This map is obviously a bijective homomorphism, hence we have $\mathbb{C}^+ \cong \mathbb{R}^+ \times \mathbb{R}^+$.

We now come to a crucial theorem which says that the concept of a quotient group is virtually equivalent to the concept of a homomorphic image.

Theorem 100.49 (First Isomorphism Theorem for Groups)**.** *Let f be a homomorphism from a group G_1 to a group G_2. Then*

$$G_1/\mathrm{Ker} f \cong \mathrm{Im} f.$$

The proof of this result can be found in any introductory text on abstract algebra. Note that $G_1/\mathrm{Ker} f$ makes sense as $\mathrm{Ker} f$ is a normal subgroup of G.

A.7. Rings

A ring is an additive finite abelian group with an extra operation, usually denoted by multiplication, such that the multiplication operation is associative and has an identity element. The addition and multiplication operations are linked via the distributive law,

$$a \cdot (b + c) = a \cdot b + a \cdot c.$$

If the multiplication operation is commutative then we say we have a commutative ring. The following are examples of rings.

- Integers under addition and multiplication of integers.
- Polynomials with coefficients in $\mathbb{Z}$, denoted $\mathbb{Z}[X]$, under polynomial addition and multiplication.
- Integers modulo a number m, denoted $\mathbb{Z}/m\mathbb{Z}$, under addition and multiplication modulo m.

Although one can consider subrings they turn out to be not so interesting. Of more interest are the ideals of the ring; these are additive subgroups $I < R$ such that

$$i \in I \text{ and } r \in R \text{ implies } i \cdot r \in I.$$

Examples of ideals in a ring are the principal ideals which are those additive subgroups generated by a single ring element. For example if $R = \mathbb{Z}$ then the principal ideals are the ideals $m\mathbb{Z}$, for each integer m.

Just as with normal subgroups and groups, where we formed the quotient group, with ideals and rings we can form the quotient ring. If we take $R = \mathbb{Z}$ and $I = m\mathbb{Z}$ for some integer m then the quotient ring is the ring $\mathbb{Z}/m\mathbb{Z}$ of integers modulo m under addition and multiplication modulo m. This leads us naturally to the Chinese Remainder Theorem.

Theorem 100.50 (CRT)**.** *Let $m = p_1^{z_1} \cdots p_t^{z_t}$ be the prime factorization of m, then the following map is a ring isomorphism*

$$f : \begin{array}{ccc} \mathbb{Z}/m\mathbb{Z} & \longrightarrow & \mathbb{Z}/p_1^{z_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_t^{z_t}\mathbb{Z} \\ x & \longmapsto & (x \pmod{p_1^{z_1}}, \ldots, x \pmod{p_t^{z_t}}). \end{array}$$

PROOF. This can be proved by induction on the number of prime factors of m. We leave the details to the interested reader. □

We shall now return to the Euler ϕ function mentioned earlier. Remember $\phi(n)$ denotes the order of the group $(\mathbb{Z}/n\mathbb{Z})*$. We would like to be able to calculate this value easily.

Lemma 100.51. *Let $m = p_1^{z_1} \cdots p_t^{z_t}$ be the prime factorization of m. Then we have*

$$\phi(m) = \phi(p_1^{z_1}) \cdots \phi(p_t^{z_t}).$$

PROOF. This follows from the Chinese Remainder Theorem, as the ring isomorphism

$$\mathbb{Z}/m\mathbb{Z} \cong \mathbb{Z}/p_1^{z_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_t^{z_t}\mathbb{Z}$$

induces a group isomorphism

$$(\mathbb{Z}/m\mathbb{Z})^* \cong (\mathbb{Z}/p_1^{z_1}\mathbb{Z})^* \times \cdots \times (\mathbb{Z}/p_t^{z_t}\mathbb{Z})^*.$$

□

To compute the Euler ϕ function all we now require is the following.

Lemma 100.52. *Let p be a prime number, then $\phi(p^e) = p^{e-1} \cdot (p-1)$.*

PROOF. There are $p^e - 1$ elements of $\mathbb{Z}$ satisfying $1 \leq k < p^e$; of these we must eliminate those of the form $k = r \cdot p$ for some r. But $1 \leq r \cdot p < p^e$ implies $1 \leq r < p^{e-1}$, hence there are $p^{e-1} - 1$ possible values of r. So we obtain

$$\phi(p^e) = (p^e - 1) - (p^{e-1} - 1)$$

from which the result follows. □

An ideal I of a ring is called prime if $x \cdot y \in I$ implies either $x \in I$ or $y \in I$. Notice that the ideals $I = m\mathbb{Z}$ of the ring $\mathbb{Z}$ are prime if and only if m is plus or minus a prime number. The prime ideals are special as if we take the quotient of a ring by a prime ideal then we obtain a field. Hence, $\mathbb{Z}/p\mathbb{Z}$ is a field. This brings us naturally to the subject of fields.

A.8. Fields

A field is essentially two abelian groups stuck together using the distributive law.

Definition 100.53 (Field)**.** *A field is an additive abelian group F, such that $F \setminus \{0\}$ also forms an abelian group with respect to another operation (which is usually written multiplicatively). The two operations, addition and multiplication, are linked via the distributive law:*

$$a \cdot (b + c) = a \cdot b + a \cdot c = (b + c) \cdot a.$$

Many fields that one encounters have infinitely many elements. Every field either contains $\mathbb{Q}$ as a subfield, in which case we say it has characteristic zero, or it contains $\mathbb{F}_p$ as a subfield in which case we say it has characteristic p. The only fields with finitely many elements have p^r elements when p is a prime. We denote such fields by $\mathbb{F}_{p^r}$; for each value of r there is only one such field up to isomorphism. Such finite fields are often called Galois fields.

Let F be a field. We denote by $F[X]$ the ring of polynomials in a single variable X with coefficients in the field F. The set $F(X)$ of rational functions in X is the set of functions of the form

$$f(X)/g(X),$$

where $f(X), g(X) \in F[X]$ and $g(X)$ is not the zero polynomial. The set $F(X)$ is a field with respect to the obvious addition and multiplication. One should note the difference in the notation of the brackets, $F[X]$ and $F(X)$.

Let f be a polynomial of degree n with coefficients in $\mathbb{F}_p$ which is irreducible. Let θ denote a root of f. Consider the set

$$\mathbb{F}_p(\theta) = \{a_0 + a_1 \cdot \theta + \cdots + a_{n-1} \cdot \theta^{n-1} : a_i \in \mathbb{F}_p\}.$$

Given two elements of $\mathbb{F}_p(\theta)$ one adds them componentwise and multiplies them as polynomials in θ but then one takes the remainder of the result on division by $f(\theta)$. The set $\mathbb{F}_p(\theta)$ is a field; there are field-theoretic isomorphisms

$$\mathbb{F}_{p^n} \cong \mathbb{F}_p(\theta) \cong \mathbb{F}_p[X]/(f),$$

where (f) represents the ideal $\{f \cdot g : g \in \mathbb{F}_p[X]\}$.

Finite Field Example 1: To be more concrete let us look at the specific example given by choosing a value of $p = 3 \pmod 4$ and $f(X) = X^2 + 1$. Now since $p = 3 \pmod 4$ the polynomial f is irreducible over $\mathbb{F}_p[X]$ and so the quotient $\mathbb{F}_p[X]/(f)$ forms a field, which is isomorphic to $\mathbb{F}_{p^2}$. Let i denote a root of the polynomial X^2+1. The field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ consists of numbers of the form $a + b \cdot i$, where a and b are integers modulo p. We add such numbers as

$$(a + b \cdot i) + (c + d \cdot i) = (a + c) + (b + d) \cdot i.$$

We multiply such numbers as

$$(a + b \cdot i) \cdot (c + d \cdot i) = (a \cdot c + (a \cdot d + b \cdot c) \cdot i + b \cdot d \cdot i^2) = (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c) \cdot i.$$

Finite Field Example 2: Let θ denote a root of the polynomial $x^3 + 2$, then an element of $\mathbb{F}_{7^3} = \mathbb{F}_7(\theta)$ can be represented by

$$a + b \cdot \theta + c \cdot \theta^2,$$

where $a, b, c \in \mathbb{F}_7$. Multiplication of two such elements gives

$$\begin{aligned}(a + b \cdot \theta + c \cdot \theta^2) \cdot (a' + b' \cdot \theta + c' \cdot \theta^2) &= a \cdot a' + \theta \cdot (a' \cdot b + b' \cdot a) + \theta^2 \cdot (a \cdot c' + b \cdot b' + c \cdot a') \\ &\quad + \theta^3 \cdot (b \cdot c' + c \cdot b') + c \cdot c' \cdot \theta^4 \\ &= (a \cdot a' - 2 \cdot b \cdot c' - 2 \cdot c \cdot b') + \theta \cdot (a' \cdot b + b' \cdot a - 2 \cdot c \cdot c') \\ &\quad + \theta^2 \cdot (a \cdot c' + b \cdot b' + c \cdot a').\end{aligned}$$

A.9. Vector Spaces

Definition 100.54 (Vector Space)**.** *Given a field K, a vector space (or a K-vector space) V is an abelian group (also denoted V) and an external operation $\cdot : K \times V \to V$ (called scalar multiplication) which satisfies the following axioms: For all $\lambda, \mu \in K$ and all $\mathbf{x}, \mathbf{y} \in V$ we have*

(1) $\lambda \cdot (\mu \cdot \mathbf{x}) = (\lambda \cdot \mu) \cdot \mathbf{x}$.
(2) $(\lambda + \mu) \cdot \mathbf{x} = \lambda \cdot \mathbf{x} + \mu \cdot \mathbf{x}$.
(3) $1_K \cdot \mathbf{x} = \mathbf{x}$.
(4) $\lambda \cdot (\mathbf{x} + \mathbf{y}) = \lambda \cdot \mathbf{x} + \lambda \cdot \mathbf{y}$.

where 1_K denotes the multiplicative identity of K.

One often calls the elements of V the vectors and the elements of K the scalars. Note that we have not defined how to (or whether we can) multiply or divide two vectors. With a general vector space we are not interested in multiplying or dividing vectors, only in multiplying them with scalars. We shall start with some examples:

- For a given field K and an integer $n \geq 1$, let $V = K^n = K \times \cdots \times K$ be the n-fold Cartesian product. This is a vector space over K with respect to the usual addition of vectors and multiplication by scalars. The special case of $n = 1$ shows that any field is a vector space over itself. When $K = \mathbb{R}$ and $n = 2$ we obtain the familiar system of geometric vectors in the plane. When $n = 3$ and $K = \mathbb{R}$ we obtain 3-dimensional vectors. Hence you can already see the power of vector spaces as they allow us to consider n-dimensional space in a concrete way.
- Let K be a field and consider the set of polynomials over K, namely $K[X]$. This is a vector space with respect to addition of polynomials and multiplication by elements of K.
- Let K be a field and E any set at all. Define V to be the set of functions $f : E \to K$. Given $f, g \in V$ and $\lambda \in K$ one can define the sum $f + g$ and scalar product λf via
$$(f + g)(x) = f(x) + g(x) \text{ and } (\lambda \cdot f)(x) = \lambda \cdot f(x).$$
We leave the reader the simple task of checking that this is a vector space.
- The set of all continuous functions $f : \mathbb{R} \to \mathbb{R}$ is a vector space over $\mathbb{R}$. This follows from the fact that if f and g are continuous then so are $f + g$ and $\lambda \cdot f$ for any $\lambda \in \mathbb{R}$. Similarly the set of all differentiable functions $f : \mathbb{R} \to \mathbb{R}$ also forms a vector space.

Vector Sub-spaces: Let V be a K-vector space and let W be a subset of V. W is said to be a vector subspace (or just subspace) of V if

(1) W is a subgroup of V with respect to addition.
(2) W is closed under scalar multiplication.

By this last condition we mean $\lambda \cdot \mathbf{x} \in W$ for all $\mathbf{x} \in W$ and all $\lambda \in K$. What this means is that a vector subspace is a subset of V which is also a vector space with respect to the same addition and multiplication laws as V. There are always two trivial subspaces of a space, namely $\{\mathbf{0}\}$ and V itself. Here are some more examples:

- $V = K^n$ and $W = \{(\xi_1, \ldots, \xi_n) \in K^n : \xi_n = 0\}$.
- $V = K^n$ and $W = \{(\xi_1, \ldots, \xi_n) \in K^n : \xi_1 + \cdots + \xi_n = 0\}$.
- $V = K[X]$ and $W = \{f \in K[X] : f = 0 \text{ or } \deg f \leq 10\}$.
- $\mathbb{C}$ is a natural vector space over $\mathbb{Q}$, and $\mathbb{R}$ is a vector subspace of $\mathbb{C}$.
- Let V denote the set of all continuous functions from $\mathbb{R}$ to $\mathbb{R}$ and W the set of all differentiable functions from $\mathbb{R}$ to $\mathbb{R}$. Then W is a vector subspace of V.

Properties of Elements of Vector Spaces: Before we go any further we need to define certain properties which sets of elements of vector spaces can possess. For the following definitions let V be a K-vector space and let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{x}$ denote elements of V.

Definition 100.55 (Linear Independence)**.** *We have the following definitions related to linear independence of vectors.*

- $\mathbf{x}$ *is said to be a linear combination of* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *if there exists scalars* $\lambda_i \in K$ *such that*
$$\mathbf{x} = \lambda_1 \cdot \mathbf{x}_1 + \cdots + \lambda_n \cdot \mathbf{x}_n.$$
- *The elements* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *are said to be linearly independent if the relation*
$$\lambda_1 \cdot \mathbf{x}_1 + \cdots + \lambda_n \cdot \mathbf{x}_n = \mathbf{0}$$
implies that $\lambda_1 = \cdots = \lambda_n = 0$. *If* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *are not linearly independent then they are said to be linearly dependent.*
- *A subset* A *of a vector space is linearly independent or free if whenever* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *are finitely many elements of* A*, they are linearly independent.*
- *A subset* A *of a vector space* V *is said to span (or generate)* V *if every element of* V *is a linear combination of finitely many elements from* A*.*
- *If there exists a finite set of vectors spanning* V *then we say that* V *is finite-dimensional.*

We now give some examples of the last concept.

- The vector space $V = K^n$ is finite-dimensional. Since if we let
$$\mathbf{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)$$
be the n-tuple with 1 in the ith place and 0 elsewhere, then V is spanned by the vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$. Note the analogy with the geometric plane.
- $\mathbb{C}$ is a finite-dimensional vector space over $\mathbb{R}$, and $\{1, \sqrt{-1}\}$ is a spanning set.
- $\mathbb{R}$ and $\mathbb{C}$ are not finite-dimensional vector spaces over $\mathbb{Q}$. This is obvious since $\mathbb{Q}$ has countably many elements, so any finite-dimensional subspace over $\mathbb{Q}$ will also have countably many elements. However it is a basic result in analysis that both $\mathbb{R}$ and $\mathbb{C}$ have uncountably many elements.

Now some examples about linear independence:

- In the vector space $V = K^n$ the n vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$ defined earlier are linearly independent.
- In the vector space $\mathbb{R}^3$ the vectors $\mathbf{x}_1 = (1, 2, 3)$, $\mathbf{x}_2 = (-1, 0, 4)$ and $\mathbf{x}_3 = (2, 5, -1)$ are linearly independent.
- On the other hand, the vectors $\mathbf{y}_1 = (2, 4, -3)$, $\mathbf{y}_2 = (1, 1, 2)$ and $\mathbf{y}_3 = (2, 8, -17)$ are linearly dependent as we have $3 \cdot \mathbf{y}_1 - 4 \cdot \mathbf{y}_2 - \mathbf{y}_3 = \mathbf{0}$.
- In the vector space (and ring) $K[X]$ over the field K the infinite set of vectors
$$\{1, X, X^2, X^3, \ldots\}$$
is linearly independent.

Dimension and Bases:

Definition 100.56 (Basis)**.** *A subset A of a vector space V which is linearly independent and spans the whole of V is called a basis.*

Given a basis, each element in V can be written in a unique way: for suppose $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is a basis and we can write $\mathbf{x}$ as a linear combination of the $\mathbf{x}_i$ in two ways i.e. $\mathbf{x} = \lambda_1 \cdot \mathbf{x}_1 + \cdots + \lambda_n \cdot \mathbf{x}_n$ and $\mathbf{x} = \mu_1 \cdot \mathbf{x}_1 + \cdots + \mu_n \cdot \mathbf{x}_n$. Then we have
$$\mathbf{0} = \mathbf{x} - \mathbf{x} = (\lambda_1 - \mu_1) \cdot \mathbf{x}_1 + \cdots + (\lambda_n - \mu_n) \cdot \mathbf{x}_n$$
and as the $\mathbf{x}_i$ are linearly independent we obtain $\lambda_i - \mu_i = 0$, i.e. $\lambda_i = \mu_i$. We have the following examples.

- The vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$ of K^n introduced earlier form a basis of K^n. This basis is called the standard basis of K^n.
- The set $\{1, i\}$ is a basis of the vector space $\mathbb{C}$ over $\mathbb{R}$.
- The infinite set $\{1, X, X^2, X^2, \ldots\}$ is a basis of the vector space $K[X]$.

By way of terminology we call the vector space $V = \{\mathbf{0}\}$ the trivial or zero vector space. All other vector spaces are called non-zero. To make the statements of the following theorems easier we shall say that the zero vector space has the basis set $\emptyset$.

Theorem 100.57. *Let V be a finite-dimensional vector space over a field K. Let C be a finite subset of V which spans V and let A be a subset of C which is linearly independent. Then V has a basis, B, such that $A \subseteq B \subseteq C$.*

Proof. We can assume that V is non-zero. Consider the collection of all subsets of C which are linearly independent and contain A. Certainly such subsets exist since A is itself an example. So choose one such subset B with as many elements as possible. By construction B is linearly independent. We now show that B spans V.

Since C spans V we only have to show that every element $\mathbf{x} \in C$ is a linear combination of elements of B. This is trivial when $\mathbf{x} \in B$ so assume that $\mathbf{x} \notin B$. Then $B' = B \cup \{\mathbf{x}\}$ is a subset

of C larger than B, whence B' is linearly dependent, by choice of B. If $\mathbf{x}_1, \ldots, \mathbf{x}_r$ are the distinct elements of B this means that there is a linear relation

$$\lambda_1 \cdot \mathbf{x}_1 + \cdots + \lambda_r \cdot \mathbf{x}_r + \lambda \cdot \mathbf{x} = \mathbf{0},$$

in which not all the scalars, λ_i, λ, are zero. In fact $\lambda \neq 0$, otherwise B would consist of linearly dependent vectors. So we may rearrange to express $\mathbf{x}$ as a linear combination of elements of B, as λ has an inverse in K. □

Corollary 100.58. *Every finite-dimensional vector space V has a basis.*

PROOF. We can assume that V is non-zero. Let C denote a finite spanning set of V and let $A = \emptyset$ and then apply the above theorem. □

The last theorem and its corollary are true if we drop the assumption of finite-dimension. However then we require much more deep machinery to prove the result. The following result is crucial to the study of vector spaces as it allows us to define the dimension of a vector space. One should think of the dimension of a vector space as the same as the dimension of the 2-D or 3-D space one is used to.

Theorem 100.59. *Suppose a vector space V contains a spanning set of m elements and a linearly independent set of n elements. Then $m \geq n$.*

PROOF. Let $A = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ span V, and let $B = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ be linearly independent and suppose that $m < n$. Hence we wish to derive a contradiction.

We successively replace the $\mathbf{x}$s by the $\mathbf{y}$s, as follows. Since A spans V, there exists scalars $\lambda_1, \ldots, \lambda_m$ such that

$$\mathbf{y}_1 = \lambda_1 \cdot \mathbf{x}_1 + \cdots + \lambda_m \cdot \mathbf{x}_m.$$

At least one of the scalars, say λ_1, is non-zero and we may express $\mathbf{x}_1$ in terms of $\mathbf{y}_1$ and $\mathbf{x}_2, \ldots, \mathbf{x}_m$. It is then clear that $A_1 = \{\mathbf{y}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$ spans V.

We repeat the process m times and conclude that $A_m = \{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ spans V. (One can formally dress this up as induction if one wants to be precise, which we will not bother with.)

By hypothesis $m < n$ and so A_m is not the whole of B and $\mathbf{y}_{m+1}$ is a linear combination of $\mathbf{y}_1, \ldots, \mathbf{y}_m$, as A_m spans V. This contradicts the fact that B is linearly independent. □

Let V be a finite-dimensional vector space. Suppose A is a basis of m elements and B a basis of n elements. By applying the above theorem twice (once to A and B and once to B and A) we deduce that $m = n$. From this we conclude the following theorem.

Theorem 100.60. *Let V be a finite-dimensional vector space. Then all bases of V have the same number of elements; we call this number the dimension of V (written $\dim V$).*

It is clear that $\dim K^n = n$. This agrees with our intuition that a vector with n components lives in an n-dimensional world, and that $\dim \mathbb{R}^3 = 3$. Note that when referring to dimension we sometimes need to be clear about the field of scalars. If we wish to emphasize the field of scalars we write $\dim_K V$. This can be important, for example if we consider the complex numbers we have

$$\dim_{\mathbb{C}} \mathbb{C} = 1, \ \dim_{\mathbb{R}} \mathbb{C} = 2, \ \dim_{\mathbb{Q}} \mathbb{C} = \infty.$$

The following results are left as exercises.

Theorem 100.61. *If V is a (non-zero) finite-dimensional vector space, of dimension n, then*

1. *Given any linearly independent subset A of V, there exists a basis B such that $A \subseteq B$.*
2. *Given any spanning set C of V, there exists a basis B such that $B \subseteq C$.*
3. *Every linearly independent set in V has $\leq n$ elements.*
4. *If a linearly independent set has exactly n elements then it is a basis.*
5. *Every spanning set has $\geq n$ elements.*
6. *If a spanning set has exactly n elements then it is a basis.*

Theorem 100.62. *Let W be a subspace of a finite-dimensional vector space V. Then* $\dim W \leq \dim V$*, with equality holding if and only if* $W = V$.

Index

3DES, *see* triple DES

A5/1 generator, 236–237
Abadi, Martín, 388
Abdalla, Michel, 330
abelian group, 4, 52, 54, 67, 70, 72, 464, 468–470
access structure, 403–406
active attack, 176
additively homomorphic, 318, 363, 364, 421
Adleman, Len, 31, 48, 202, 203, 216
Adleman–Huang algorithm, 31
Advanced Encryption Algorithm, *see* AES
AES, 107, 131, 164, 241–243, 250–254, 261, 266, 278, 289, 291, 336
affine point, 68
Agrawal, Manindra, 31
AKS Algorithm, 27, 31
algebraic normal form, 233
alternating-step generator, 235–236
American National Standards Institute, 245
anomalous, 77
ANSI, *see* American National Standards Institute
approximation factor, 83, 85
arithmetic circuit, 366, 440, 445
associative, 4, 459, 461, 468
asymmetric cryptosystems, 119
authenticated encryption, 266
authenticated key agreement, 387, 394–398
automorphism, 10
avalanche effect, 246, 251, 291

Babai's algorithm, 89
Baby-Step/Giant-Step, 57–59, 245, 336
BAN logic, 388–392
Banaszczyk transference theorem, 83, 358
basis, 80, 81, 472
basis matrix, 80, 81
Baudot code, 182, 184, 185, 192
Bayes' Theorem, 22, 168
BDD, *see* Bounded-Distance Decoding
Bellare, Mihir, 321, 330, 335
Berlekamp–Massey algorithm, 233
Berlekamp–Welch algorithm, 411–413
Bertoni, Guido, 289
Bertrand, Gustave, 141, 147
bigrams, 120
bijective, 456, 467
binary circuit, 440, 441, 445
binary Euclidean algorithm, 13, 105, 108
binary exponentiation, 97
binding, 418–421
birthday paradox, 23–24, 59, 272
bit security, 219–221
BLAKE, 289
Blake-Wilson, Simon, 386
Blake-Wilson–Menezes protocol, 386–387, 396–398
block cipher, 8, 66, 124, 127, 180–182, 236, 238, 241–255, 262, 263, 276, 285, 287–289, 300
Bombe, 140, 150–156, 159
Boneh, Dan, 308
bootstrapping, 366
Bounded-Distance Decoding, 87–89, 360
Burrows, Michael, 376, 388

CA, *see* certificate authority
Caesar cipher, 120
CAMELLIA, 242
Carmichael numbers, 30
Cartesian product, 453
CBC-MAC, 285–287
CCA, 176, 204–214, 222, 242, 320, 325, 326
certificate, 371–375, 414, 415
  authority, 371–375, 414
  binding, 371
  chain, 373
  implicit, 374–375
  revocation, 374
  revocation list, 374
CFRAC, *see* continued fraction method
ChaCha, 289
characteristic, 7, 9–11, 68, 70, 73–77, 107, 108, 469
Chaum–Pedersen protocol, 432–433
Chinese Remainder Theorem, 3, 11, 15–17, 20, 21, 25, 36, 54, 57, 100, 111, 298, 304, 305, 310, 314, 319, 468, 469
chord-tangent process, 69, 70
chosen ciphertext attack, *see* CCA
chosen plaintext attack, *see* CPA
cillies, 142
cipher, 119
ciphertext, 119
closest vector problem, 85–89

Cocks, Clifford, 202
code-word, 407–410, 412, 413
coding theory, 407
codomain, 340, 455
collision resistance, 271–275
Colossus, 192, 193
commitment scheme, 417–421, 435
commutative, 4, 459, 461, 468
commutative ring, 468
complexity class
  $\mathcal{CZK}$, 428, 429
  $\mathcal{IP}$, 428, 429
  $\mathcal{NP}$, 350–351, 428, 430
  $\mathcal{NP}$-complete, 351
  $\mathcal{NP}$-hard, 351
  $\mathcal{PSPACE}$, 428, 429
  $\mathcal{P}$, 27, 31, 350, 351
complexity theory, 349–353
compression function, 276–279, 282, 283, 285, 288
computationally secure, 163, 418
concealing, 418–421
conditional entropy, 171, 172
conditional probability, 22, 165, 171
confusion, 251
conjugate, 463
connection polynomial, 228
continued fraction, 85, 306–308
continued fraction method, 39
Coppersmith's Theorem, 90–93, 305, 306, 308, 309
Coppersmith, Don, 90, 306
coprime, 6
correlation attack, 234, 235
coset, 464, 465
Couveignes, Jean-Marc, 48
CPA, 131, 176, 204, 206, 207, 209, 210, 213, 214, 222, 296, 320
Cramer, Ronald, 326, 346, 433
Cramer–Shoup encryption, 342–344
Cramer–Shoup signature, 344–346
crib, 142
CRL, *see* certificate revocation list
cross-certification, 373
CRT, *see* chinese remainder theorem
CTR Mode, 285
CVP, *see* closest vector problem
cycle, 457
cyclic group, 5, 462

Daemen, Joan, 250, 289
Damgård, Ivan, 276, 433
data encapsulation mechanism, 213, 218, 267, 287, 324–330
Data Encryption Algorithm, *see* DES
Data Encryption Standard, *see* DES
data integrity, 216
DDH, *see* Decision Diffie–Hellman problem
DEA, *see* DES
decipherment, 119
Decision Diffie–Hellman problem, 53–54, 317–318, 330, 331, 343, 352, 353, 356
decision game, 34, 199
decision problem, 349, 350
decryption, 119
DEM, *see* data encapsulation mechanism
DES, 131, 164, 241–252, 254
DHAES, *see* DHIES-KEM
DHIES-KEM, 329–332
DHP, *see* Diffie–Hellman problem
difference, 453
differential cryptanalysis, 243, 245
Diffie, Whitfield, 202, 203, 216, 383
Diffie–Hellman key exchange, 383–386
Diffie–Hellman problem, 52–54, 316–317, 331, 384
Diffie–Hellman protocol, 383–385, 387
diffusion, 251
digital certificate, *see* certificate
digital signature, 215–218, 333–342, 344–346, 382, 387
Digital Signature Algorithm, *see* DSA
Digital Signature Standard, *see* DSA
dimension, 473
discrete logarithm, 5, 219–220
discrete logarithm problem, 42, 51–66, 77, 201, 301, 315, 324, 333, 335, 336, 341, 383, 384, 387
discriminant, 68, 82
disjoint, 458
disjunctive normal form, 405–406
distributive, 4, 468, 469
DLP, *see* discrete logarithm problem
DNF, *see* disjunctive normal form
Dolev–Yao model, 377
domain, 455
domain parameters, 316, 342, 362, 384
DSA, 95, 309, 335–338, 340, 344, 346, 374, 375
DSS, *see* DSA
Durfee, Glen, 308

EC-DH, 383, 385
EC-DSA, 335–338, 340, 374
ECIES, *see* DHIES-KEM
ECM, *see* elliptic curve method
ECPP, *see* elliptic curve primality proving
ElGamal encryption, 27, 95, 164, 313, 315–319, 324, 335, 343, 362, 421
elliptic curve, 8, 27, 31, 38, 51, 52, 67–78, 99, 101, 107, 316, 330, 335–338, 383, 385
elliptic curve method, 38, 39
elliptic curve primality proving, 31
Ellis, James, 202, 203
encipherment, 119
Encrypt-and-MAC, 268
Encrypt-then-MAC, 266–268, 293, 330
encryption, 119
Enigma, 117, 133–160, 164, 182
entropy, 169–174
ephemeral key, 316
equivalence class, 454
equivalence relation, 67, 69, 453, 454, 468

error-correcting code, 407
Euclidean algorithm, 12–14, 108, 306
Euclidean division, 12, 103, 463
EUF, *see* existential unforgeability
EUF-CMA, 217, 218, 283, 284, 292, 301, 333, 335, 338, 339, 430, 432
Euler ϕ function, 6, 462, 469
exhaustive search, 180
existential forgery, 217, 218, 299, 301, 334, 340
existential unforgeability, 217, 218
extended Euclidean algorithm, 14, 52, 74, 295, 296

factor group, 465
factorbase, 42
factoring, 31–49, 52, 301
fault analysis, 310
feedback shift register, 227
Feistel cipher, 244–246, 250
Feistel network, 321
Feistel, Horst, 244
female, 148, 149
Fermat Test, 29, 30
Fermat's Little Theorem, 7, 28, 40
FHE, *see* fully homomorphic encryption
Fiat–Shamir heuristic, 431
field, 7, 251, 469
filter generator, 235
finite field, 7–11, 27, 31, 51, 55, 58, 64, 66, 67, 72–74, 77, 99, 107, 108, 111, 229, 316, 335, 338, 385, 469
finite-dimensional, 471
fixed field, 10
Flexible RSA Problem, 344, 345
Floyd's cycle-finding algorithm, 59–61
forking lemma, 339–341
forward secrecy, 383, 394
Franklin, Matthew, 306
Franklin–Reiter attack, 305–306
frequency analysis, 122
Frobenius endomorphism, 73
Frobenius map, 10, 72
Fujisaki, Eiichiro, 323, 324
Fujisaki–Okamoto transform, 324, 329
fully homomorphic encryption, 319, 363–366
function, 455
Function Field Sieve, 52, 64
fundamental parallelepiped, 82, 359

Galois field, 9, 469
Gap Diffie–Hellman problem, 330–332, 397, 398
garbled circuit, 440–445
Gauss, Carl Friedrich, 27
Gaussian Heuristic, 82, 90
gcd, *see* greatest common divisor
GCHQ, 202
Geffe generator, 234, 235
Generalized Riemann Hypothesis, 30, 31, 351
generator, 5, 462
generic group model, 338
Gentry, Craig, 319, 364
geometric series assumption, 84
Gillogly, James, 159
Goldwasser, Shafi, 31, 313, 344
Goldwasser–Micali encryption, 313–315, 318
Gram–Schmidt, 80, 83, 84
graph, 349
graph isomorphism, 425–429
greatest common divisor, 6, 8, 12, 13, 128
group, 4, 28, 52, 461–468
Grøstl, 289
GSA, *see* geometric series assumption
GSM, 236

Hamming weight, 96
hard predicate, 219
hash function, 271–293, 300, 301, 320, 321, 324, 333–335, 338, 340–342
Hasse's Theorem, 77
Hellman, Martin, 54, 202, 203, 216, 351, 383
Hermite Normal Form, 90
hiding, 418
HMAC, 278, 282–285, 287
homomorphic property, 297
homomorphism, 54, 466, 467
honest-but-curious, 439, 440, 448
Howgrave-Graham, Nick, 90
Huang, Ming-Deh, 31
hybrid encryption, 268, 324–332, 370, 382
hyperelliptic curve, 31
Håstad's attack, 305–306
Håstad, Johan, 305

ideal, 47, 48, 468
identity, 459, 461
image, 467
IND security, 205–214, 217, 225, 227, 238, 242, 255, 258, 259, 293, 313–332
IND-CCA, 206–214, 263, 266–269, 292, 293, 297, 315, 322, 324–329, 332
IND-CPA, 206–209, 213, 214, 258, 259, 261, 263–267, 292, 297, 313–315, 317–319, 324, 421, 422
IND-PASS, 206–209, 212–214, 225, 227, 238, 255, 258, 259, 266
independent, 22
index, 464
index of coincidence, 159
index-calculus, 64
Indian exponentiation, 97
indistinguishability of encryptions, *see* IND security
information-theoretic security, 164, 205, 418
injective, 167, 456, 467
inner product, 79
International Organization for Standardization, 245
intersection, 453
inverse, 456, 460, 461
irreducible, 8
ISO, *see* International Organization for Standardization
isomorphic, 9, 467

isomorphism, 68, 467, 470

Jacobi symbol, 18
Jensen's inequality, 170
JH, 289
joint entropy, 171
joint probability, 22
Joux, Antoine, 66

Karatsuba multiplication, 102, 103, 110
Kasiski test, 128
KASUMI, 236
Kayal, Neeraj, 31
KDF, *see* key derivation function
Keccak, 289–291
KEM, *see* key encapsulation mechanism
Kerberos, 381, 388
Kerckhoffs' principle, 179, 198
kernel, 467
key agreement, 382, 388, 390–398
key derivation function, 274, 284–285, 287–288, 292, 321, 329, 330, 332, 384, 386
key distribution, 169, 180
key encapsulation mechanism, 324–332
key equivocation, 172
key exchange, 370, 383–388
key schedule, 243, 250
key transport, 382–384, 388
Kilian, Joe, 31
knapsack problem, 349–351, 353–355

Lagrange interpolation, 408, 409, 413, 415, 446
Lagrange's Theorem, 7, 8, 17, 28, 38, 108, 296, 319, 464, 465
Las Vegas algorithm, 59, 301
lattice, 79–93, 305, 355–360
  q-ary, 89–90, 360
  dual, 83
  minimum, 81
learning with errors, 360–366
  Ring-LWE, 361–366
Legendre symbol, 17, 18, 48, 314
Lehmer, Derrick, 31
Lenstra, Arjen, 83
Lenstra, Hendrik W., 83
LFSR, *see* linear feedback shift register
linear combination, 471
linear complexity, 182, 233, 234
linear complexity profile, 233
linear feedback shift register, 8, 227–238
linearly dependent, 471
linearly independent, 80, 471, 472
LLL algorithm, 83–86, 92, 355, 356
LLL reduced basis, 83, 84
long Weierstrass form, 67
Lorenz, 117, 182–193
Lovász, László, 83
LTE, 236
Lucifer, 244
lunchtime attacks, 212

LWE, *see* learning with errors

MAC, 215, *see* message authentication code
MAC-then-Encrypt, 268–269
malicious adversary, 440
malleable encryption, 214
man-in-the-middle attack, 378, 385, 395
MARS, 242
matching conversation, 393
MD-4 family, 278–282, *see also* SHA family
  MD-4, 279–280
  MD-5, 278, 279, 289
MDC, *see* manipulation detection code
Menezes, Alfred, 386, 387
Merkle, Ralph, 276, 351
Merkle–Damgård construction, 276–285, 288
Merkle–Hellman encryption, 354, 355
message authentication code, 215–218, 266–268, 278, 282–288, 292, 293
message recovery, 216, 299, 300
Micali, Silvio, 313, 344
Miller–Rabin Test, 30, 31
millionaires problem, 439
Minkowski's Theorem, 82
Minkowski, Hermann, 82
modes of operation, 254–265, 276
  CBC Mode, 254, 257–266, 268, 285–287
  CCM Mode, 266
  CFB Mode, 255, 263–266
  CTR Mode, 255, 264–266, 274, 287
  ECB Mode, 254–258, 264–266, 300
  GCM Mode, 266
  OCB Mode, 266
  OFB Mode, 255, 262–266
modular arithmetic, 454
modulus, 3
monotone access structure, 404, 405, 407
monotone structure, 404
Montgomery arithmetic, 103, 105–108
Montgomery multiplication, 107
Montgomery reduction, 105–107
Montgomery representation, 103–106
Montgomery, Peter, 101
MPC, *see* multi-party computation
MQV protocol, 386–388
multi-party computation, 439–449
multiplicatively homomorphic, 318, 363, 415

National Institute of Standards and Technology, 242, 280, 289
natural language, 173, 174
Needham, Roger, 388
Needham–Schroeder protocol, 376–382, 388
negligible function, 34
NFS, *see* Number Field Sieve
NIST, *see* National Institute of Standards and Technology
NMAC, 283–284, 287
non-deterministic polynomial time, 350

non-malleability, 213–214
nonce, 375, 380, 382, 389, 390
nonce-based encryption, 212–213, 258, 259, 264, 265
normal subgroup, 463–465
NSA, 244
Number Field Sieve, 38–40, 44–49, 52, 65, 336
Nyberg–Rueppel signature, 341–342

OAEP, 321, *see* RSA-OAEP
oblivious transfer, 421–423, 444
Okamoto, Tatsuaki, 323, 324
one-time pad, 164, 169, 180
one-time signature, 218
one-way function, 201–203, 219, 429
operation, 459
optimal normal bases, 107
Optimized Asymmetric Encryption Padding, *see* RSA-OAEP
order, 462
orthogonal, 79, 80
ot-IND-CCA, 213, 214
OW-CCA, 209, 256, 265, 297, 300, 318, 319
OW-CPA, 204, 209, 256, 262, 263, 265, 296, 298, 317
OW-PASS, 204, 209, 265
OWF, *see* one-way function

padding, 275–276
Paillier encryption, 313, 318–319, 363
Paillier, Pascal, 318
partial key exposure, 309–310, 335
partial order relation, 454
PASS, 204, 209, 210, 212, 214, 225, 227, 238, 255
passive attack, 176, 180, 204, 313
Pedersen commitment scheme, 421, 433, 435, 436
Peeters, Michaël, 289
pentanomial, 108
perfect cipher, 175
perfect security, 164, 166, 205
perfect zero-knowledge, 427
period of a sequence, 227
permutation, 131, 135, 136, 139, 144, 151, 288, 456–458, 460, 462
permutation cipher, 131
PKI, *see* public key infrastructure
plaintext, 119
plaintext aware, 214, 324
plugboard, 136–141, 147, 148, 150, 153–157, 159
Pocklington, Henry, 31
Pohlig, Stephen, 54
Pohlig–Hellman algorithm, 54–58
point at infinity, 67
point compression, 75
point multiplication, 99
Pointcheval, David, 323, 339
Pollard's Kangaroo method, 62
Pollard's Lambda method, 62–63
Pollard's Rho method, 52, 59–61, 77
  parallel version, 63–64
Pollard, John, 40, 59
polynomial security, 205
preimage resistant, 271
PRF, *see* psuedo-random function
primality-proving algorithm, 27
prime ideal, 47, 48
prime number, 27–31, 463
Prime Number Theorem, 27
primitive, 228
primitive element, 11
principal ideal, 468
private key, 179
probabilistic signature scheme, 334
probability, 21
probability distribution, 21
probable prime, 29
product group, 468
Project Athena, 381
projective plane, 67
projective point, 67, 68
PRP, *see* psuedo-random permutation
PRSS, *see* secret sharing, pseudo-random
PRZS, *see* secret sharing, zero sharing
pseudo-prime, 29
pseudo-random function, 197–201, 207, 221, 225–227, 272, 274, 278, 285, 287
pseudo-random permutation, 200–201, 241, 242, 244, 256, 257, 259, 262–265, 278, 288
pseudo-random secret sharing, *see* secret sharing, pseudo-random
pseudo-random zero sharing, *see* secret sharing, zero sharing
pseudo-squares, 19, 313
public key, 179
public key cryptography, 32, 179, 202–203, 370
public key cryptosystems, 119
public key encryption, 202, 295–297, 313–332
public key infrastructure, 371, 374
public key signature, 216–218, 299–301, 333–342, 344–346, 431–432

QS, *see* quadratic sieve
Qu, Minghua, 387
quadratic reciprocity, 17
quadratic sieve, 38, 39, 45
QUADRES, 32, 34–35, 37, 53, 313–315
qualifying set, 403–406
quantum computer, 353, 360
quotient group, 465

Różycki, Jerzy, 140
RA, *see* registration authority
Rabin encryption, 27, 298–299, 319
Rabin, Michael, 27, 298
random oracle model, 221–222, 275, 320–322, 324, 329–334, 338–342, 344, 345, 384, 422
random self-reduction, 352–353
random variable, 21
random walk, 59, 60, 62–64
RC4, 238, 239

RC5, 238
RC6, 238, 242
redundancy, 174
redundancy function, 341
Reed–Solomon code, 407–413, 448
reflector, 134, 136, 141, 145
registration authority, 373
Reiter, Michael, 306
Rejewski, Marian, 140
relation, 453, 468
relatively prime, 6
replicated secret sharing, 406, *see* secret sharing, replicated
residue classes, 455
Rijmen, Vincent, 250
Rijndael, 242, 250
ring, 5, 251, 468, 469, 472
RIPEMD-160, 278, 279
Rivest, Ron, 202, 203, 216, 238, 344
Rogaway, Phil, 272, 321, 330, 335
ROM, *see* random oracle model
RoR security, 210–212, 214
round function, 242
round key, 243
RSA, 15, 32, 33, 37, 38, 41, 51, 77, 95, 100, 101, 164, 202, 203, 219–221, 295–310, 313, 316, 318, 319, 321–324, 329–330, 333–336, 341, 344–346, 352, 353, 374, 414–416
RSA encryption, 27, 295–297, 303, 304, 306, 363
RSA problem, 33, 38, 202, 220–221, 295–297, 301, 322, 329, 331, 333, 334, 345, 352, 353, 356
RSA signature, 299–301, 305, 333–336
  distributed, 414–416
RSA-FDH, 300, 301, 333–335
RSA-KEM, 329–331, 333
RSA-OAEP, 321–323, 329, 334
RSA-PSS, 333–335, 344

S-Box, 246–248, 252, 254
safe primes, 41
Saxena, Nitin, 31
Schmidt, Hans-Thilo, 141
Schnorr signature, 338–341, 344, 429–431
Schnorr's identification protocol, 429–431
Schoenmakers, Berry, 433
second preimage resistant, 271, 274
secret key cryptosystems, 119
secret sharing, 371, 403–416, 440, 445–449
  Ito–Nishizeki–Saito, 405–406
  pseudo-random sharing, 413–414, 448
  pseudo-random zero sharing, 414, 448
  replicated, 406–407
  Shamir, *see* Shamir secret sharing
selective forgery, 217
semantic security, 205–206
Serpent, 242
session key, 370
sEUF, *see* strong existential unforgeability
sEUF-CMA, 218
SHA family
  SHA-0, 280, 281
  SHA-1, 278–281, 289
  SHA-2, 278, 279, 281–282, 289
  SHA-256, 279, 281–282, 284
  SHA-384, 279
  SHA-512, 279
SHA-3, 279, 289–291
Shamir secret sharing, 407, 412–415, 436, 440, 445–449
Shamir's trick, 100, 101
Shamir, Adi, 202, 203, 216, 407
Shanks' Algorithm, 18, 36
Shanks, Daniel, 18, 57
Shannon's Theorem, 167–169, 205
Shannon, Claude, 167, 169
Sherlock Holmes, 124
shift cipher, 120, 121, 123, 124, 126–128, 164, 168, 173
short integer solution, 90, 357–360
short Weierstrass form, 69
shortest independent vectors problem, 358–360
shortest vector problem, 85–86, 90, 357–360
Shoup, Victor, 326, 346, 415
shrinking generator, 236
Sieve of Eratosthenes, 28, 45
Sigma protocol, 429–436
  "Or" proofs, 433–436
signature scheme, 301
signed sliding window method, 99
signed window method, 99
Sinkov statistic, 138, 159
SIS, *see* short integer solution
Skein, 289
sliding window method, 98, 99
small inverse problem, 308
smart card, 371
Smith Normal Form, 42
smooth number, 39
smoothness bound, 42
somewhat homomorphic encryption, 364–366
SP-network, 250
span, 471
sponge construction, 288–293
spurious keys, 173, 175
SQRROOT, 32, 33, 35–37, 53, 298
square and multiply, 97
standard basis, 472
standard model, 221, 338, 342
Station-to-Station protocol, 386
statistical distance, 122, 123, 128, 129, 428
Stern, Jacques, 323, 339
stream cipher, 8, 121, 124, 127, 181, 182, 225–240, 243, 255, 262, 263, 274, 289, 292
strong existential unforgeability, 218, 266
STS, *see* station-to-station protocol
subfield, 9
subgroup, 462, 463
substitution cipher, 123, 124, 126, 133, 164, 173, 175

super-increasing knapsack, 353
supersingular, 77
surjective, 456, 467
SVP, *see* shortest vector problem
symmetric cryptosystems, 119
symmetric encryption, 276
symmetric key, 179

Threefish, 289
threshold access structure, 404, 406, 407
Tiltman, John, 188
timestamp, 376, 381, 382, 390
TLS, 238
total order relation, 454
trace of Frobenius, 72
traffic analysis, 180
trapdoor one-way function, 203, 329
trapdoor one-way permutation, 321
trial division, 28, 38, 39
trigrams, 120
trinomial, 108
triple DES, 242, 245–246
Trivium, 238
trusted third party, 372, 375
TTP, *see* trusted third party
Turing, Alan, 150, 152, 154
Tutte, William, 188
Twofish, 242

UMTS, 236
unconditionally secure, 164
unicity distance, 173, 175, 176
union, 453
universal composability framework, 369

Van Assche, Gilles, 289
Vanstone, Scott, 387
vector space, 470–472
vector subspace, 471
Vernam cipher, 169, 181
Vigenère cipher, 127, 164

Weierstrass equation, 67
Welchman, Gordon, 152
Wide-Mouth Frog protocol, 376, 379, 390–392
Wiener's attack, 306–308
Wiener, Mike, 306
Williamson, Malcolm, 203
window method, 97–99
witness, 29

Yao circuit, *see* garbled circuit
Yao, Andrew, 440, 443

zero-knowledge, 339, 416, 425–436
  completeness, 429
  computational, 428
  knowledge extractor, 430–432
  non-interactive, 432
  proof of knowledge, 430
  soundness, 429, 430
  special soundness, 430, 433
  statistical, 428
Zygalski sheet, 148, 149
Zygalski, Henryk, 140, 148